



INSTITUTO POLITÉCNICO  
DO CÁVADO E DO AVE  
ESCOLA SUPERIOR DE TECNOLOGIA

# C# Essencial

**Paradigmas de Programação I**

**Paradigmas de Programação II**

**Programação I**

**Programação II**

**Material de Apoio**

## **Autores**

- António Tavares
- Eva Oliveira
- João Carlos Silva
- Célio Carvalho
- Luís Ferreira
- Manuela Cunha
- Marco Costa



INSTITUTO POLITÉCNICO  
DO CÁVADO E DO AVE  
ESCOLA SUPERIOR DE TECNOLOGIA

# Índice

<b>Parte I – Configurações e C# Essencial</b>	<b>12</b>
C# .....	12
Requisitos para programar em C#: .....	13
Compilar (sem o Visual Studio) .....	14
Compilar (com o Visual Studio) .....	15
Fundamentos: .....	16
Estrutura base de um programa em C# .....	16
Comentários .....	16
Documentar código em XML .....	17
Variáveis (Camel Notation) e Tipos de Dados .....	20
Constantes .....	21
Operadores .....	21
Exemplo .....	22
Estruturas de controlo .....	25
Exercícios: .....	27
Conceitos OO: .....	29
Resumo: .....	29
Recomendações CLS: .....	30
Referências .....	30
 <b>Parte II – Arrays</b>	 <b>31</b>
Arrays .....	31
Arrays uni-dimensionais: .....	31
Manipulação de Arrays - Exemplos .....	32
Exemplo Completo .....	34
Arrays multidimensionais .....	36
Manipulação de Arrays Multidimensionais – Exemplos .....	37

Jagged arrays .....	37
Manipulação de Jagged Arrays - Exemplos .....	38
Exemplo completo: .....	39
Exercícios: .....	39
A seguir: .....	40
Resumo: .....	40
Recomendações CLS: .....	40
Referências .....	40
 <b>Parte III – Arrays (II)</b>	 <b>41</b>
Revisão: .....	41
Operações avançadas sobre Arrays .....	43
Ordenação .....	43
Procura .....	45
Passagem de Arrays como parâmetros .....	46
Métodos com número variável de parâmetros ( <i>params</i> ) .....	46
Comparando “ <i>params</i> ” com “...” em JAVA .....	50
Exemplo completo .....	52
Exercícios .....	55
Referências .....	55
 <b>Parte IV – Recursividade</b>	 <b>56</b>
Recursividade: .....	56
Definição: .....	56
Exemplos: .....	56
Síntese .....	60
Características .....	60
Recursividade versus Ciclos .....	61
Vantagens e Desvantagens .....	61

Conclusão.....	61
Referências .....	62
 <b>Parte V – Programação Orientada a Objectos</b>	 <b>63</b>
Conceito de classe .....	63
Objectos.....	63
Métodos.....	64
Instâncias .....	65
Modificadores de acesso .....	65
Membros <i>static</i> de uma classe .....	67
Construtores .....	67
Propriedades.....	69
Definição de propriedades <i>set</i> e <i>get</i> .....	69
Utilização de Propriedades .....	70
Override de Métodos.....	71
Operadores <i>is</i> e <i>as</i> .....	73
Exemplo completo .....	74
Referências .....	76
 <b>Parte VI – Collections</b>	 <b>78</b>
ArrayList.....	78
Stack .....	79
Queue.....	80
HashTable .....	80
SortedList.....	82
Exercício 1:.....	83
Exercício 2:.....	84
Referências .....	84

<b>Parte VII – Herança</b>	<b>86</b>
Alguma Terminologia .....	86
Herança em POO .....	86
Herança em C++ .....	90
Herança em JAVA .....	90
Herança em C# .....	91
Exemplo: Empregado → Gestor → Supervisor .....	91
Reutilização .....	93
Mais conceitos .....	95
Membros <i>Protected</i> .....	95
Classes <i>sealed</i> .....	96
Exemplos completos .....	96
Polimorfismo .....	103
Reescrever Métodos – <i>override</i> .....	105
Operador <i>new</i> .....	106
Classes Abstratas .....	107
Interfaces .....	109
Múltiplos Interfaces .....	110
Herança de Interfaces .....	114
Interfaces vs Classes abstratas .....	115
Exercícios .....	116
Referências .....	118
 <b>Parte VIII – Exceções</b>	 <b>119</b>
Introdução .....	120
Erros .....	120
Exceções .....	121
Tipos de exceções .....	123
Criação de novos tipos de exceções .....	127

Libertação de recursos.....	128
Lançamento de exceções.....	129
Exercícios .....	130
Referências .....	132
 <b>Parte IX – Windows Forms</b>	 <b>133</b>
Manipulação de Forms .....	133
Interface MDI.....	133
Definição e inserção de um menu de opções .....	133
Adicionar uma nova janela filha .....	135
Fechar a janela filha activa .....	136
Organização das janelas filhas .....	137
Utilização do componente TabControl .....	137
O componente <i>ErrorProvider</i> .....	138
Os componentes indispensáveis.....	141
Exercícios .....	143
Referências .....	144
 <b>Parte X – Manipulação de Ficheiros e Serialização</b>	 <b>146</b>
Considerações.....	146
<i>Patterns</i> para manipulação de Ficheiros .....	146
Ficheiros de Texto .....	146
Ficheiro Binários .....	149
Outros métodos disponíveis em <i>File.IO</i> .....	151
Manipular ficheiros em Windows Forms .....	152
Exemplos.....	154
Referências .....	160
 <b>Parte XI – ADO.NET</b>	 <b>161</b>

Introdução .....	161
Acesso a bases de dados no modo tradicional .....	162
Acesso a bases de dados no modo desconectado.....	165
Referências .....	171
 <b>Parte XII – Persistência</b>	 <b>172</b>
Considerações.....	172
Mapeamento de Objectos para Bases de Dados .....	172
Shadow Information .....	174
Mapeamento de Herança de classes .....	174
Mapear Relações .....	177
Implementar Persistência .....	178
Bases de Dados Orientadas a Objectos .....	178
Boa conduta em Programação .....	179
Exemplo .....	179
Classe DAO .....	180
Classe Pessoa .....	186
Classe Curso .....	187
Classe do Interface da aplicação:.....	189
Referências .....	194
 <b>Parte XIII – Delegates e Eventos</b>	 <b>195</b>
 <b>Parte XIV – Ofuscação e SVN</b>	 <b>196</b>



## Figuras

Figura 1 – Arquitectura .NET .....	12
Figura 2 – Java versus C# .....	13
Figura 3 – Criar um novo projecto no Visual Studio .....	15
Figura 4 – Configurar Documentação XML (I).....	18
Figura 5 – Configurar Documentação XML (II).....	18
Figura 6 – Array uni-dimensional .....	31
Figura 7 – Resultado da cópia de arrays (I) .....	34
Figura 8 – Resultado da cópia de arrays (II) .....	34
Figura 9 – Resultado do Exemplo completo sobre arrays simples .....	35
Figura 10 – Resultado do Exemplo sobre <i>Jagged Arrays</i> .....	39
Figura 11 – Bubble Sort .....	44
Figura 12 – Classes vs Instâncias.....	87
Figura 13 – Herança de classes .....	88
Figura 14 – Diagrama UML de Classes .....	88
Figura 15 - Especialização de uma classe .....	103
Figura 16 - Exercício sobre Herança de classes.....	117
Figura 17 – Formulário SDI .....	134
Figura 18 - Formulário MDI .....	134
Figura 19 - Formulário MDI com dois formulários SDI.....	136
Figura 20 – Componente TabControl .....	137
Figura 21 – Componente ErrorProvider .....	138
Figura 22 – Componente RichTextBox.....	139
Figura 23 - Componentes Button, LabelTextBox, ComboBoxListBox, Checked ListBox, Radio Button e TreeView .....	142
Figura 24 - Componentes PictureBoxDate, TimePicker e MonthCalendar.....	143
Figura 25 – Botão "Browse" .....	152
Figura 26 – Controlo OpenFileDialog.....	153
Figura 27 – Controlo FolderBrowserDialog .....	154

Figura 28 – Exercício.....	156
Figura 29 – De Modelo Classe para Modelo Físico .....	173
Figura 30 - Mapeamento de informação Extra.....	174
Figura 31 - Preservar Hierarquias de classes.....	175
Figura 32- Mapeamento de Relações .....	177
Figura 33- Diagrama de Classes do Exemplo .....	180
Figura 34 - Interface da aplicação exemplo.....	189

## **Apresentação**

Este documento constitui Material de Apoio para as disciplinas de Linguagens de Programação e Programação Orientada a Objectos. Resultou da compilação do trabalho de vários docentes.

## Parte I – Configurações e C# Essencial

Autor: lufer

---

Este documento pretende constituir uma compilação resumida do que é essencial para começar a trabalhar com o C#.

Numa primeira parte serão abordados os conceitos do C# como uma linguagem normal estruturada. Numa segunda parte, abordaremos a perspectiva do C# como linguagem Orientada a Objectos.

A abordagem do C# será sempre apoiada em exemplos de código, de forma simples e objectiva.

### C#

Embora se trate de uma nova (recente) linguagem, quem tiver alguma experiência em programação em praticamente qualquer linguagem, não terá grande dificuldade em atingir o essencial do C#. A dificuldade poderá advir do facto do C# ser um paradigma Orientado a Objectos com algumas novidades curiosas.

O C# é uma linguagem de Programação Orientação a Objectos que deriva essencialmente dos princípios que estão por trás da do C, C++ e JAVA. Foi criada para ser suportada pela arquitectura .NET.

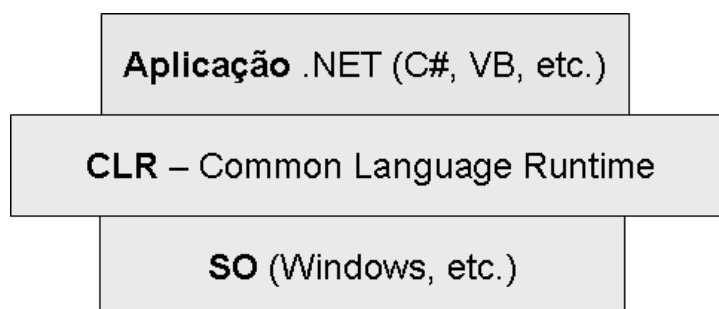


Figura 1 – Arquitectura .NET

Talvez uma forma possível de perceber o que se passa com a compilação de um programa C# seja comparar com o que se passa com um programa em JAVA. O conceito de máquina virtual repete-se mas agora chama-se *CLR – Common Language Runtime*. O *Bytecode* do Java é visto aqui como *MSIL – Microsoft Intermediate Language*. Durante a execução do programa, código MSIL é convertido em código nativo através do *JITER – Just In Time Compiler*.

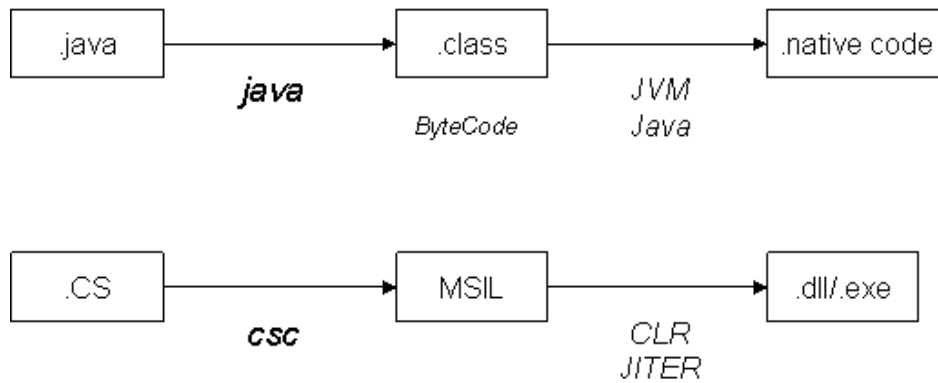


Figura 2 – Java versus C#

Um pequeno exemplo escrito em C# (leia-se c-sharp)

```

using System;
namespace Glorioso{
class Exemplo
{
public static void Main(String[] args)
{
Console.WriteLine("Msg1: " + "Benfica");
Console.WriteLine("Msg2: " + "é o maior");
}
}
  
```

Código 1 – Exemplo de Programa em C#

### Requisitos para programar em C#:

#### Compilador:

No mínimo é essencial ter instalado o .NET Framework<sup>1</sup>. Não é obrigatório ter instalado o Visual Studio .NET.

Editor para escrever o código:

- Visual Studio .NET
- Visual Studio C# Express

<sup>1</sup> <http://www.microsoft.com/downloads/Search.aspx?displaylang=en>

- (<http://msdn2.microsoft.com/en-us/express/future/default.aspx>)
- SharpDeveloper:
- <http://www.icsharpcode.com/OpenSource/SD/>
- Notepad++; Ultraedit, outros.

### Compilar (sem o Visual Studio)

1. Com qualquer editor de texto, escrever o código do *Programa* modelo, apresentado antes. Atribua ao ficheiro o nome *Exemplo.cs*.
2. Numa shell do DOS, compilar com o compilador de C# – *csc*<sup>2</sup> (Csharp Compiler)

```
c:\csc Exemplo.cs
```

3. Será gerado um ficheiro *Exemplo.exe*. Basta executar.
4. Executar com

```
c:\Exemplo
```

#### Nota:

Com o csc pode utilizar, entre muitas outras, as seguintes opções:

```
C:\csc [/help] [/out:] [/doc:] [/t:library] [/main:]
```

/help: mostra todas as opções do csc

/out: definir nome para o programa compilado

/doc: definir nome para o ficheiro xml de documentação

/t:library: gerar um .dll em vez de um .exe

/main: definir qual a classe Main a executar

---

<sup>2</sup> O local onde está instalado o comando csc.exe deverá constar na variável de ambiente PATH

## Compilar (com o Visual Studio)

A exposição deste tópico está relativamente dependente da versão do Visual Studio instalada. O apresentado refere-se à versão 2005.

1. Criar um novo projecto
2. Definir como Aplicação Consola Windows

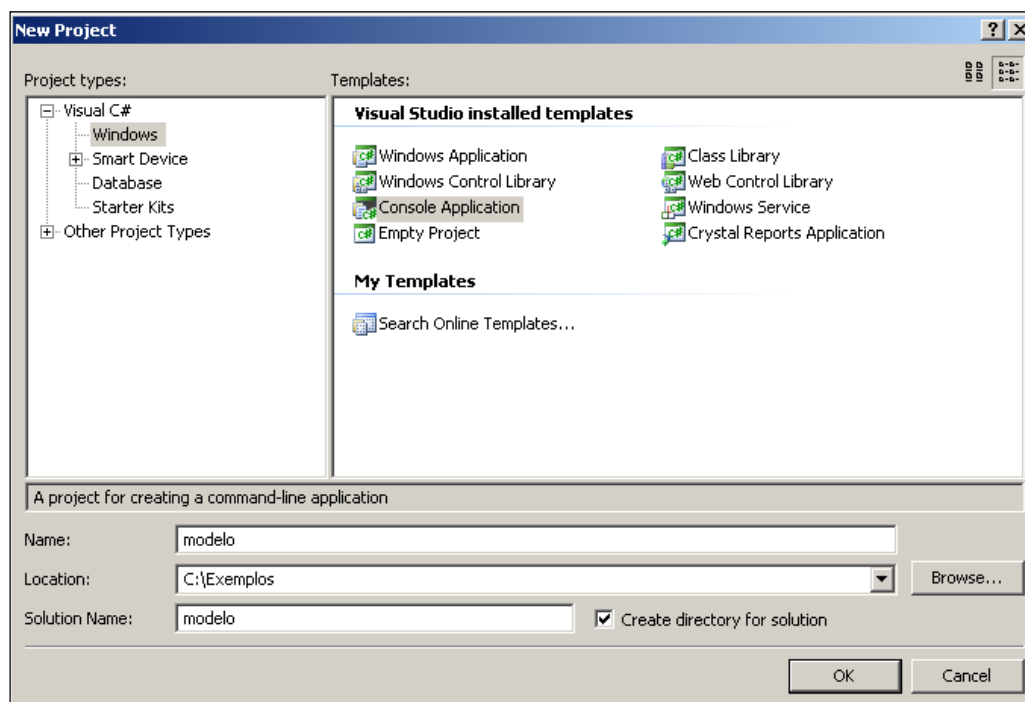
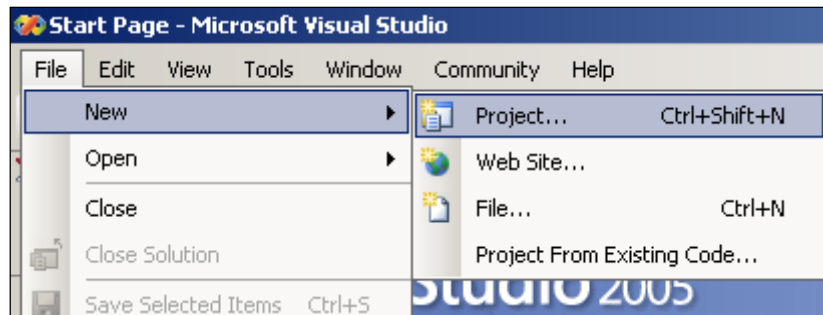


Figura 3 – Criar um novo projecto no Visual Studio

3. Escrever o código
4. Compilar com F6 (Build Solution)
5. Executar com F5 ou Ctrl+F5

## Fundamentos:

Os fundamentos associados à linguagem C# serão apresentados com exemplos simples e objectivos.

### Estrutura base de um programa em C#

```
using System;           //opcional
namespace nomeNamespace{ //opcional
class nomeClasse
{
public static void Main(String[] args)
{
}
}
```

<i>using</i>	Permite ter acesso a todas as classes definidas no namespace <i>System</i> (embora na realidade não o seja, pode ser visto como os packages do JAVA). Não é obrigatório.
<i>namespace</i>	Usado para evitar conflito de nomes. Num programa em C# não é possível haver duas classes com o mesmo nome. Um conjunto de classes relacionadas deve ser colocadas no mesmo namespace. Dentro de um <i>namespace</i> podem ser definidos: classes, eventos, excepções, delegates, e namespaces internos. Não é obrigatório.
<i>class</i>	São conjuntos de dados e métodos que descrevem uma entidade. Num programa C# deve existir pelo menos uma classe que contém um método <i>Main()</i> .
<i>Main()</i>	Primeiro método a utilizar na execução do programa. Num mesmo namespace podem existir mais do que um método <i>Main()</i> .

### Comentários

```
//isto é um comentário

/*
    Isto também é um comentário
*/
```



**Nota:**

Não confundir com JAVA, no qual, para documentar código para ser processado pelo JAVADOC, se utilizava a sintaxe:

```
/**
 *
 *
 */
```

**Documentar código em XML**

Permite documentar todo o código que se escreve. Importante para o próprio programador assim como futuros utilizadores do mesmo código.

Inicia-se com `///`

Algumas Tags possíveis:

- `///<summary>`
- `///<remarks>`
- `///<param name="args"></param>`
- `///<example>`
- `///<code>`
- `///<exception cref="SampleException">`
- `///<returns>`
- `///<seealso cref="GiveMemberListHTMLHelp"/>`

Para gerar o documento XML com os documentários, pode ser utilizado directamente no Visual Studio ou a linha de comandos com o `csc`:

**No Visual Studio 2005**

1. Aceder às Propriedades (*Properties*) do actual projecto

2. Aceder ao tabulador *Build* (Figura 5)
3. Definir o caminho para colocar o documento xml (por omissão em *bin\Debug\*).

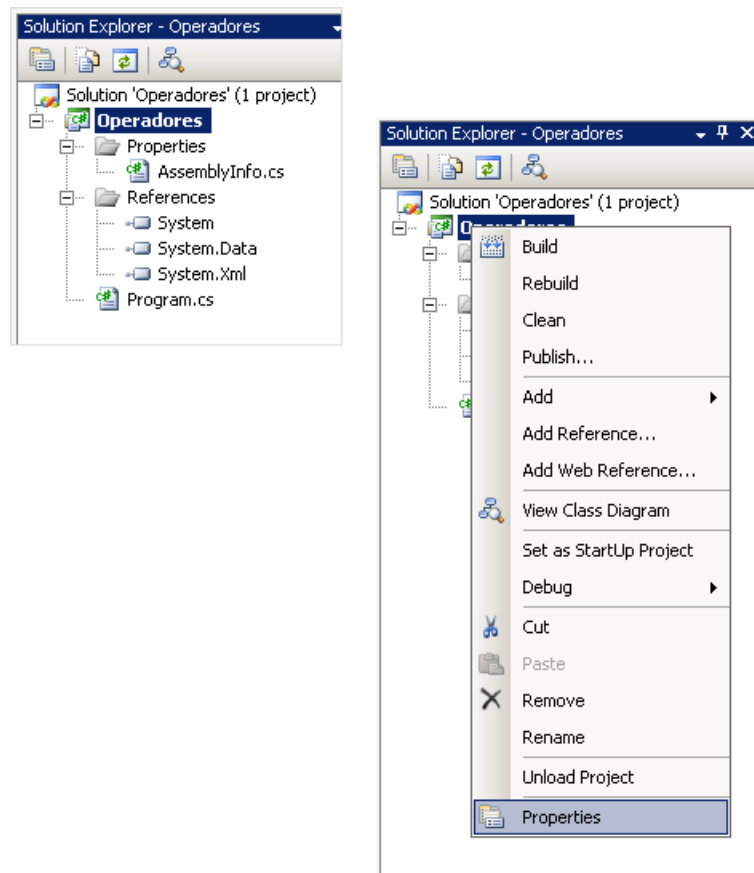


Figura 4 – Configurar Documentação XML (I)

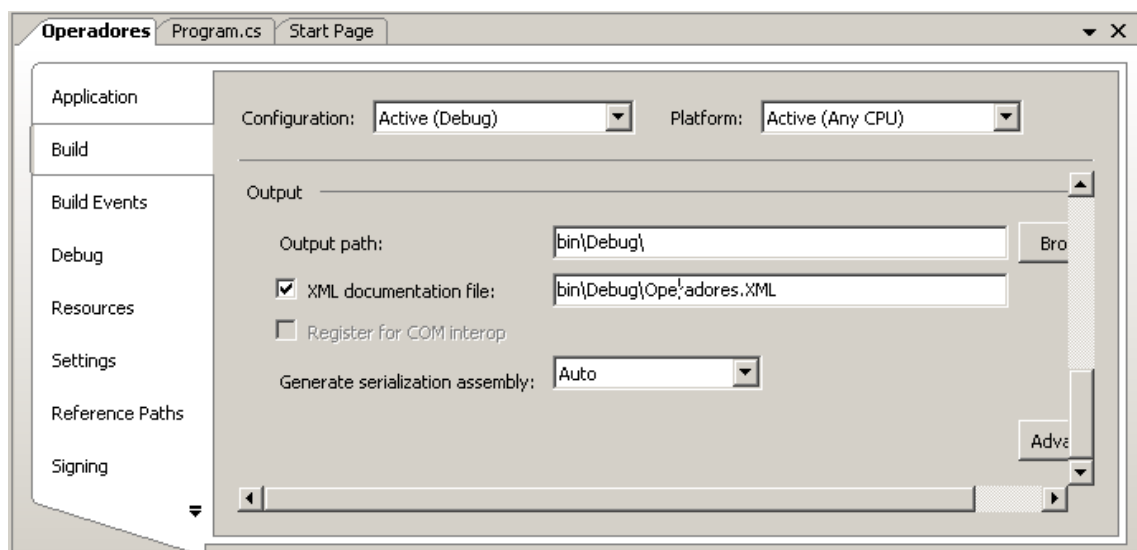


Figura 5 – Configurar Documentação XML (II)

Primeiro deve-se indicar o caminho onde colocar o ficheiro xml:

**Na linha de comandos:**

Tal como notificado antes neste documento, basta utilizar a opção **/doc:** do comando **csc**:

```
csc /doc:Manual.xml program.cs
```

Neste caso será gerado o documento *Manual.xml*. Se lhe for aplicada uma stylesheet adequada é possível gerar um help em HTML ou até mesmo um *chm* (ficheiro de help típico do Windows).

De seguida apresentam-se as tags XML mais utilizadas (é possível criar outras tags):

TAG	PROPÓSITO
<summary> ... </summary>	Breve descrição de uma classe, método ou propriedade.
<remarks> ... </remarks>	Descrição mais detalhada.
<para> ... </para>	Permite delinear parágrafos dentro da tag <remarks>
<list type="..." ... </list>	Permite usar marcadores para formatar uma descrição. Os tipos de marcadores podem ser "bullet", "number" e "table".
<example> ... </example>	Disponibilizar um exemplo de como usar um método, propriedade ou outro membro da biblioteca.
<code> ... </code>	Para indicar que o texto incluído é código da aplicação.
<see cref="member"/>	Indica uma referência para outro membro ou campo. O compilador verifica se o membro realmente existe.
<exception> ... </exception>	Para fazer a descrição de uma exceção.
<permission> ... </permission>	Para documentar a acessibilidade.
<param name="name"> ... </param>	Para documentar um parâmetro de um método.

<code>&lt;returns&gt; ... &lt;/returns&gt;</code>	Para documentar o valor devolvido por um método.
<code>&lt;value&gt; ... &lt;/value&gt;</code>	Para descrever uma propriedade.

### Variáveis (Camel Notation) e Tipos de Dados

No C# convencionou-se (respeitando a CLS) que:

- Nome das variáveis deve ser feita em notação *Camel*
- Nome dos métodos deve ser feita em notação *Pascal*
- A notação *Hungarian* deve ser evitada

Camel Notation: Palavras iniciadas com letra minúscula. Restantes palavras iniciadas com letra maiúscula (*Camel Case*)

*myName, average, bestValue*

Pascal Notation: Palavras iniciadas com letra maiúscula

*Average(), Add(), Main()*

Hungarian Notation: prefixo da palavra indica o tipo de dados

*intAge, strName, floatAverage*

*TipoDados nomeVariavel [= valorInicial];*

`int aux = 20;`

`long l = 12;`

`short tot;`

`double myArea = 2.5;`

`float x = 3.7f;`

*Conversão explícita de tipos (cast)*

`int aux = (int) (a+1);`

`float origem = 3.764f;`

`int destino = (int) origem;`

*Outros tipos:*

`bool b; // true ou false`

`char ch = '\n';`

```
string s="Benfica";
```

*Operações sobre strings:*

Concatenação	"luís" + "ferreira"
Tamanho	nome.Length
Comparação	"lufer"=="lufer"
Parte de String	nome.Substring(0,15)

## Constantes

Palavra reservada *const*. O valor é inalterável.

```
const double PI = 3.1415926535;
```

```
double raio = 12.3;
```

```
double perímetro = 2 * PI * raio;
```

## Operadores

Existem vários tipos de operadores. Abordamos somente os Aritméticos, Relacionais e Lógicos.

### ***Aritméticos***

```
int x=2,y=6;
```

```
x++; //incrementa x de uma unidade
```

```
x--; //decrementa x de uma unidade
```

```
--x; //decrementa x de uma unidade
```

```
++x; //incrementa x de uma unidade
```

```
x = x*y;
```

```
x = x-y;
```

```
x = x+y;
```

```
x = x/y;
```

```
x = x%y;
```

```
x += y;
```

```
x *= y;
```

```
x -= y;
```

```
x /= y;
```

```
x %= y;
```

### **Relacionais**

```
if (x == y){}; //igualdade
```

```
if (x != y){}; //diferença
```

```
if (x > y){}; //maior
```

```
if (x < y){}; //menor
```

```
if (x >= y){}; //maior ou igual
```

```
if (x <= y){}; //menor ou igual
```

### **Lógicos**

```
bool b=false, c=true;
```

```
b=(b && c); // "and"; b=false
```

```
b=(b || c); // "or"; b=true
```

```
b=!b // negação; b=true
```

### **Exemplo**

O exemplo seguinte mostra a aplicabilidade de variáveis, operadores e estruturas de controlo

```
using System;
namespace Operadores
{
    /// <summary>
    /// Variáveis, Operadores e Estruturas de controlo
    /// </summary>
    class Program
    {
        /// <summary>
        /// Este programa mostra a utilização de variáveis e operadores
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            int x=2,y=6;
            int soma, diff, mult, div, rest;
```

```

soma = x + y;
diff = x - y;
mult = y * y;
div = x / y;
rest = x % y; //resto da divisão

//Apresenta o resultado na forma x+y=z
Console.WriteLine("{0}+{1}={2}", x, y, soma);
Console.WriteLine("{0}-{1}={2}", x, y, diff);
Console.WriteLine("{0}*{1}={2}", x, y, mult);
Console.WriteLine("{0}/{1}={2}", x, y, div);
Console.WriteLine("{0}%{1}={2}", x, y, rest);

//Apresenta o resultado na forma x += y
Console.WriteLine("{0}+={1}={2}", x, y, (x += y));
Console.WriteLine("{0}-={1}={2}", x, y, (x -= y));
Console.WriteLine("{0}*={1}={2}", x, y, (x *= y));
Console.WriteLine("{0}/={1}={2}", x, y, (x /= y));
Console.WriteLine("{0}%={1}={2}", x, y, (x %= y));

x = 3;
Console.WriteLine("++x=" + ++x); //escreve 4; x=4
Console.WriteLine("x++=" + x++); //escreve 4; x=5
Console.WriteLine("--x=" + --x); //escreve 4; x=4
Console.WriteLine("x--={0}; x={1}", x--, x); //escreve 4; x=3;
Console.WriteLine("x++-x={0}", ((x++)-x)); //escreve -1; x=4
Console.WriteLine("++x-x={0}", ++x - x); //escreve 0; x=5

//operadores relacionais
//if...else
x = 3; y = x;
if (x < y)
{
    Console.WriteLine("x é menor que y");
}
else if (x > y)
{
    Console.WriteLine("y é menor que x");
}
else
{

```

```

        Console.WriteLine("x é igual a y");
    }

    //operador c ? a1 : a2
    Console.WriteLine((x < y) ? "x é menor que y" : "y menor ou igual a x");

    //operadores lógicos

    //operador == e !
    if (!(x == y)){
        Console.WriteLine("x é diferente de y");
    }
    else {
        Console.WriteLine("x é igual de y");
    }

    //operadores && (and) e || (or)

    bool a=false, b=!a;
    Console.WriteLine("{0} && {1}={2}", a, b, a&& b);
    Console.WriteLine("{0} || {1}={2}", a, b, a || b);
}
}
}

```

Resultado da execução do programa

```

C:\ F:\I386\system32\cmd.exe
2+6=8
2-6=-4
2*6=36
2/6=0
2%6=2
2+=6=8
8-=6=2
2*=6=12
12/=6=2
2%=6=2
++x=4
x++=4
--x=4
x--=4; x=3
x++-x=-1
++x-x=0
x é igual a y
y menor ou igual a x
x é igual de y
False && True=False
False || True=True
Prima qualquer tecla para continuar . . . _

```



## Estruturas de controlo

### Condicionais

#### if-else

```

if (x < y)
{
    Console.WriteLine("x é menor que y");
}
else
    if (x > y)
    {
        Console.WriteLine("y é menor que x");
    }
    else
    {
        Console.WriteLine("x é igual a y");
    }

```

#### switch

```

int valor=2;
switch (valor)
{
    case 1: Console.WriteLine("Um");
            break;
    case 2:
    case 3: Console.WriteLine("Dois ou Três");
            break;
    default: Console.WriteLine("Nenhum deles");
}
break;

```

### Ciclos

Todos os ciclos apresentados mostram o resultado: 2,4,6,8

#### while

```

int k = 2;
while (k<10) {
    Console.WriteLine("k= {0}", k);
    k+=2;
}

```

#### do-while

```

int k = 2;
do{
    Console.WriteLine("k= {0}", k);
    k+=2;
}while (k<10)

```

#### for

Deve ser utilizado sempre que se conhece o número exacto de vezes que o ciclo irá ser executado.

```

for (int k = 2; k <= 7; k += 2)
{
    Console.WriteLine("k= {0}", k);
}

```

```
}
```

Como cada uma das seções do "for" é opcional, é possível utilizar o ciclo for seguinte forma.

```
for(;;)
```

```
for(;k<=7;k+=2)
```

```
for(;k<=7;)
```

```
for(int k=2;;)
```

### foreach

Utilizado essencialmente para analisar o conteúdo de arrays. Será analisado com mais detalhe aquando a abordagem de arrays.

```
int[] elementos={10,20,30,40};

foreach (int i in elementos) {

    Console.WriteLine("i= {0}", i);

}
```

### Controlo dos ciclos: *break* e *continue*

Pode acontecer que não seja necessário executar o ciclo no número de vezes previsto, ou seja, o ciclo pode ser interrompido. Pode também interessar alterar a sequência da execução do ciclo. Isto faz-se com as instruções: *break* e *continue*.

A instrução *break* pára o ciclo.

A instrução *continue* força a nova iteração do ciclo.

Analisemos o ciclo definido anteriormente mas alterado com um *continue*:

```
int k = 2;
while (k<10) {
    if (k%2==0) {k++;continue;};
    Console.WriteLine("k= {0}", k);
    k+=2;
}
```

Neste caso, o resultado apresentado seria 3,5,7,9. Procure analisar porquê!

Agora com *break*:

```
int k = 2;
while (k<10) {
    if (k%2==0) {break;};
    Console.WriteLine("k= {0}", k);
    k+=2;
}
```

Qual o valor agora apresentado?

### Exercícios:

1. Crie um Projecto no Visual Studio C# com as seguintes particularidades:

- Chama-se Ficha\_1
- Documente em XML devidamente a classe que criou
- Deve apresentar escrito no ecran o enunciado da Questão 1
- Declare (correctamente) o equivalente às seguintes variáveis:
  - Inteiros minha\_idade, data\_nascimento;
  - String meu\_nome
- Inicialize as variáveis com os valores que pretender
- Mostre no ecran o nome e a idade da pessoa

2. Continue o exercício 1. Mas agora:

- Leia do stdin o nome da pessoa e a respectiva data de nascimento
- Calcule a sua idade
- Mostre o nome da pessoa e a respectiva idade

3. Corrija eventuais erros no seguinte programa em C#

```
//Aulas de LP
/by lufer
namespace Aula
{
    /// <summary>
    /// Tratamento de Tipos de Dados, Operadores e Expressões
    /// </summary>
    class Program
    {
        /// <summary>
        /// 
        /// </summary>
        /// <param name="args"></param>
        static void Main(string[] args)
        {
            System.Console.WriteLine ("Nome completo");
            Console.Writeline("Nome abreviado");

            Int i = 10;    //32 bits = 4 bytes
            long l = 12234; //64 bits
            Float f = (int)0.234;
            double d = 0,45;
            char c = "a";
        }
    }
}
```

```

bool b = 1;
string s = Console.Write("Numero : "); Console.ReadLine();

//Conversão de tipos
i = int.Parse(string);
Console.WriteLine(s + " - " - i + " - " - i.ToString());
    }
}
}

```

4. Analise se estão ou não correctos os seguintes excertos de código em C#. Verifique o resultado e no caso de existir algum erro ou *warning*, tente-o corrigir.

a.

```

int x=6;

double y;

float z = 2.345F;

Console.WriteLine(x + "-" + y);

```

b.

```

int x;

if (x<2) Console.WriteLine("Maior");

```

c.

```

int x = 2;

Console.WriteLine("x+1=" + (x + 1));           Console.WriteLine("x+1=" + (++x));
Console.WriteLine("x+1=" + (x++));

Console.WriteLine(x > 10);

```

d.

```

string name = "Benfica"

Console.Write (name[3]);

```

5. Analise o seguinte excerto de código em C# e diga qual o resultado esperado:

```

Console.Write("Numero : ");
string s = Console.ReadLine(); i = int.Parse(s);
Console.WriteLine(s + " - " + i + " - " + i.ToString());

bool b = true;
Console.WriteLine("b & {0}!={1}",!b,! (b && false));

int x=7,y;
Console.WriteLine("Valor de x é {0}",(x%2==0)?"Par":"Impar");

```

```

if (++x % 2 == 0)
    y = x++;
else
    y = --x;
Console.WriteLine("X=" + x + " Y=" + y);

```

6. Escreva um pequeno programa em C# capaz de ler do teclado um conjunto de valores inteiros. Atenda às seguintes regras:
  - O projecto deve chamar-se *folha2\_exe3*
  - a leitura termina quando for inserido o valor 0 (zero)
  - o programa deve apresentar:
    - quantos números foram inseridos
    - quantos números pares foram inseridos
    - o maior número inserido
    - o menor número inserido
7. Execute o mesmo programa que fez na alínea 3 mas agora com as restantes estruturas de controlo: for(;;); do..while, while().

### Conceitos OO:

O C# segue praticamente todo o padrão OO do JAVA, com algumas particularidades que convém anotar. Nas partes seguintes deste documento abordaremos estes conceitos.

### Resumo:

- C# é *case-sensitive* . *int* é diferente de *Int*

Método Main() represente o “arranque” da aplicação

Espaços, tabuladores e CR são ignorados

É importante comentar em XML todo o código

O nome do ficheiro não tem que possuir o mesmo nome que a classe que contém o Main()

Podem existir múltiplos métodos Main() num programa

Blocos de código representam-se com chavetas {}

- A palavra-chave *using* permite utilizar bibliotecas externas (*namespaces*)
- O *namespace* não é obrigatório

O método Main não necessita de ter argumentos.

Todas as instruções terminam com “;”.

Existem essencialmente três tipos de operadores: aritméticos, lógicos e condicionais

As variáveis têm de ser declaradas antes de serem utilizadas

- *Existem várias estruturas de controlo. Condicionais: if..else e switch; Ciclos: for(); do..while, while.*
- *As estruturas de controlo podem ser controladas com break, continue or goto (não abordado neste documento);*

### Recomendações CLS:

- Nomear variáveis com notação *CamelCase* (myName). Começar com letra minúscula.
- Nomear Métodos com notação *Pascal* (Main)
- Não usar notação *Hungarian* (strName)
- Nomes devem começar com caracteres não dígitos (\_ é um carácter)
- Nomes de variáveis não devem usar “\_”
- Nomes de variáveis não devem diferir somente pelo facto de ser maiúsculas ou minúsculas. Por exemplo, não declarar variáveis myName e MyName.

### Referências

<http://www.softsteel.co.uk/tutorials/cSharp/>

<http://www.csharp-station.com/Tutorial.aspx>

<http://csharpcomputing.com/Tutorials/TOC.htm>

<http://msdn2.microsoft.com/en-us/vcsharp/aa336804.aspx>

C# School – Programmers Heaven

## Parte II – Arrays

Autor: lufer

---

Nesta segunda parte deste documento *C# Essencial*, serão abordadas algumas estruturas de dados, nomeadamente arrays uni emultidimensionais e Jagged arrays.

### Arrays

Um array é uma estrutura de dados que possibilita o armazenamento de dados de um determinado tipo. Cada valor armazenado numa unidade desse array, é acedida ou indexada por um indexador do tipo inteiro, correspondente a essa posição.

#### Arrays uni-dimensionais:

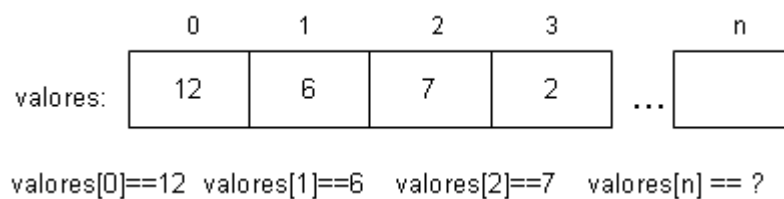


Figura 6 – Array uni-dimensional

Array que comporta uma única dimensão de dados.

#### Sintaxe:

*tipoDados* Array[] nomeArray = **new** *TipoDados*Array[dimensãoArray]

*Classe* Array[] nomeArray = **new** *Classe*[dimensãoArray]

**Declarar e Activar** (reservar espaço com o operador **new()**)

- `double[] alturas = new double[10];`
- `alturas[3]=12.3;`
- `string[] nomes = new string[3];`
- `nomes[0] = "lufer";`

**Declarar, inicializar e omitir o tamanho do array.**

```
string[] cp = {"4900","4500","4750"};
```

```
int[] valores = {1,2,3,4,5,6,7,8};
```

**Manipulação de Arrays - Exemplos**

Declarações:

```
//SINTAXE:
//type[] array_name = new type[size];
//type[] array_name = {val1,val2,...,valN};

// declarar
int[] valores;
valores = new int[20];
//int[] valores = new int[20]; Equivalente

//declarar e inicializar sem operador new()
int[] notas ={12,13,10,15};

// array de strings
string[] nomes ={ "ola", "ole" };
```

Operadores e Operações:

```
//dimensão de um array: Length()
Console.WriteLine("Tamanho do array notas=" + notas.Length);

// Inserir valores num array
for (int i = 0; i < valores.Length; i++)
    valores[i] = i;
```



Percorrer um array:

```
// Inserir valores num array
for (int i = 0; i < valores.Length; i++)
    valores[i] = i;

// Mostrar todos os valores de um array
int j = 0;
while (j < valores.Length)
{
    Console.WriteLine("aux[{0}]={1}", j, valores[j]);
    j++;
}
// Mostrar todos os valores de um array com foreach
int x = 0;
foreach (int i in valores)
{
    Console.WriteLine("valores[{0}]={1}", x++, i);
}
```

Copiar arrays:

O array destino deverá ter a dimensão suficiente para conter o array origem.

*Exemplo 1:*

```
// Copiar um array: copiar array notas para array novo
int[] novo=new int[notas.Length+3];
notas.CopyTo(novo, 0); //copia a partir da posição "0" do array notas

// Mostrar todos os valores de um array com foreach
//int x = 0;
x = 0;
foreach (int i in novo)
{
    Console.WriteLine("novo[{0}]={1}", x++, i);
}
```

*Resultado:*

Repare-se que o array novo fica somente com as quatro posições ocupadas, correspondente a todos os valores que existiam no array notas

```

novo[0]=12
novo[1]=13
novo[2]=10
novo[3]=15
novo[4]=0
novo[5]=0
novo[6]=0

```

Figura 7 – Resultado da cópia de arrays (I)

Exemplo 2:

```

notas.CopyTo(novo, 3); //insere a partir da posição "3" do array novo

// Mostrar todos os valores de um array com foreach
//int
x = 0;
foreach (int i in novo)
{
    Console.WriteLine("novo[{0}]={1}", x++, i);
}

```

*Resultado*

Neste caso é feita uma cópia do array notas mas inseridas a partir da posição 3 do novo array.

```

novo[0]=12
novo[1]=13
novo[2]=10
novo[3]=12
novo[4]=13
novo[5]=10
novo[6]=15

```

Figura 8 – Resultado da cópia de arrays (II)

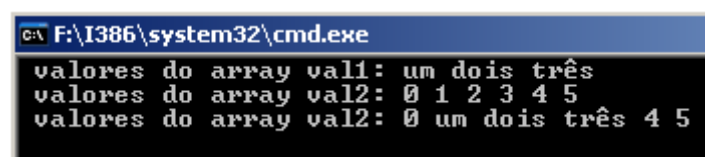
## Exemplo Completo

```

public class CopyToArray
{
    public static void Main()
    {
        //val1 e val2 são arrays com valores do tipo object
        object[] val1 = {"um", "dois", "três"};
        object[] val2 = {0, 1, 2, 3, 4, 5};
        Console.Write (" valores do array val1: ");
        foreach(object o in val1)
        {
            Console.Write (" {0} ", o);
        }
        Console.Write ("\n valores do array val2: ");
        foreach (object o in val2)
        {
            Console.Write (" {0} ", o);
        }
        val1.CopyTo (val2, 1);
        Console.Write ("\n valores do array val2: ");
        foreach (object o in val2)
        {
            Console.Write (" {0} ", o);
        }
        Console.WriteLine();
    }
}

```

### Resultado



```

C:\F:\I386\system32\cmd.exe
valores do array val1: um dois três
valores do array val2: 0 1 2 3 4 5
valores do array val2: 0 um dois três 4 5

```

Figura 9 – Resultado do Exemplo completo sobre arrays simples

### Nota

Uma exceção (*Exception*) ocorre sempre que surge um erro na execução de um programa. Contudo é possível evitar que o programa pare de forma anormal na sequência desse erro. Chama-se a isto *suportar Exceções*. Quando a referência utilizada num array é inválida (ex:  $i > \text{array.Length}$  ou  $i < 0$ ), o C# gera a exceção ***IndexOutOfRangeException***. Na continuidade deste documento será analisada a forma como em C# lida com *Exceptions*.

## Arrays multidimensionais

Arrays com mais do que uma dimensão. Os arrays bidimensionais são vulgarmente conhecidos por Matrizes.

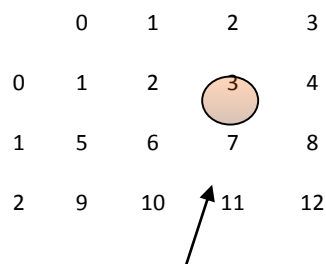
### Sintaxe:

*tipoDadosArray[,]* nomeArray = **new** *TipoDadosArray[dimensão1,dimensão2]*

### Declarações

```
int[,] valores = new int[3,4];    //array multidimensional 3 por 4
```

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12



```
valores[1,2]=7;
```

```
int [,] valores = {{2,3},{3,4}};
```

*Array tri-dimensional:*

```
int[,,] Salarios;
```

## Manipulação de Arrays Multidimensionais – Exemplos

```
// Arrays Multidimensionais
//SINTAXE:
//type[,] array_name = new type[size1,size2];
//type[,] array_name = {{val1,val2,...,valN},{val1,val2,...,valN}};

//novidade no C#...em Java: int[][]
int[,] multvals = new int[3,4];
multvals[0,0] = 2;
multvals[0,1] = 3;

int[,] idades = { { 1, 1, 2,3 }, { 7, 8 ,5,0}, { 9,13,24,1 } };           //idades[3,4]
/* Na prática significa:
 * |1|1|2|3
 * |7|8|5|0
 * |9|13|24|1
 * */

string[,] emails ={ { "lufer","eoliveira"}, {"mcunha","jcsilva"} }; //emails[2,2]
```

### Operadores e Operações:

```
string[,] emails ={ { "lufer","eoliveira"}, {"mcunha","jcsilva"} }; //emails[2,2]

// Tamanhos das duas dimensões
Console.WriteLine("Linhas={0}", emails.GetLength(0).ToString()); //3
Console.WriteLine("Colunas={0}", emails.GetLength(1).ToString()); //3

// Mostrar todos os valores de um array multidimensional
j = 0;
for (; j < emails.GetLength(0); j++)
{
    for (int i = 0; i < emails.GetLength(1); i++)
        Console.WriteLine(emails[i, j]);
}
```

## Jagged arrays

Caso particular de um array multidimensional. Tratam-se de arrays com múltiplas dimensões de tamanho variável. Resumindo, tratam-se de arrays de arrays.

### Sintaxe:

```
type[][] array_name = new type[size][];
```

**Declarações:**

```
int[][] vals = new int[3][];

vals[0] = new int[2];

vals[1] = new int[3];

vals[2] = new int[4];
```

```

           0    1    2    3
0
1
2
```

**Manipulação de Jagged Arrays - Exemplos**

Antes de utilizar um jagged array é necessário dimensionar todas as dimensões.

```
//Jagged arrays
//type[][] array_name = new type[size][];

int[][] vals = new int[3][]; // vals: array de array de inteiros
vals[0] = new int[2];
vals[1] = new int[3];
vals[2] = new int[4];
vals[2][0] = 1;
//vals[0][2]=3;           //erro: indice excede dimensão do array
```

Manipular:

```
//"jagged" array: array of(array of int)
int[][] nums = new int[3][];
nums[0] = new int[] { 1, 2, 3 };
nums[1] = new int[] { 1, 2, 3, 4, 5, 6 };
nums[2] = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

Console.WriteLine("Array[0][1]={0}", nums[0][1]); // valor 2
Console.WriteLine("Arrays={0}", nums.Length);    //número de arrays: 3
```

```
// array de array bidimensional
int[,] varios = new int[3][,]
{
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
};

// mostra o valor da célula [1,0] do primeiro array
Console.WriteLine("{0}", varios[0][1, 0]);
```

### Exemplo completo:

```
public class ArrayTest
{
    static void Main()
    {
        // Declara um jagged array (2x?)
        int[][] arr = new int[2][];

        // Inicializa o array
        arr[0] = new int[5] { 1, 3, 5, 7, 9 };
        arr[1] = new int[4] { 2, 4, 6, 8 };

        // Mostra o array
        for (int i = 0; i < arr.Length; i++)
        {
            Console.Write("Linha({0}): ", i);

            for (int j = 0; j < arr[i].Length; j++)
            {
                Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ? "" : " ");
            }
            Console.WriteLine();
        }
        Console.WriteLine("OK.");
    }
}
```

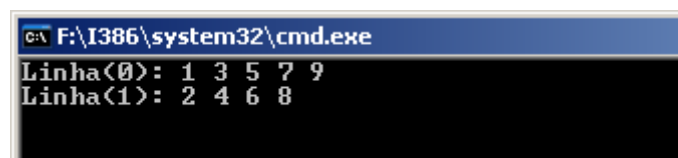


Figura 10 – Resultado do Exemplo sobre *Jagged Arrays*

### Exercícios:

Defina um método

```
public static void Copy(Array, int, Array, int, int);
```

capaz de copiar o conteúdo entre dois arrays, partindo de uma posição do array inicial e copiar para o array final a partir de uma posição específica

**A seguir:**

Nas partes seguintes deste documento abordaremos os arrays com Classes e respectivos métodos associados. Abordaremos também operações mais complexas como procuras e processos de ordenação.

**Resumo:**

Os arrays em C# divergem algo do JAVA e C/C++.

O símbolo “[]” surge junto ao tipo de dados (ex. int[] valores)

- Ao inicializar um array não se coloca o operador *new()* tal como é feito no JAVA

**Recomendações CLS:**

- Respeitar as regras de nomeação de variáveis do tipo Array (*Camel Case*)

**Referências**

<http://www.softsteel.co.uk/tutorials/cSharp/>

<http://www.csharp-station.com/Tutorial.aspx>

<http://csharpcomputing.com/Tutorials/TOC.htm>

[http://msdn2.microsoft.com/en-us/library/9b9dty7d\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/9b9dty7d(VS.80).aspx)

[http://msdn2.microsoft.com/en-us/library/2s05feca\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/2s05feca(VS.80).aspx)

C# School – Programmers Heaven



## Parte III – Arrays (II)

Autor: lufer

---

Nesta terceira parte deste documento C# Essencial, serão abordadas mais algumas questões sobre arrays, nomeadamente procura e processos de ordenação. Serão também analisados a forma como arrays podem ser utilizados por métodos.

### Revisão:

Como vimos na Parte II deste documento, um array é uma estrutura de dados que possibilita o armazenamento de dados de um determinado tipo. Cada valor armazenado numa unidade desse array, é acedida ou indexada por um indexador do tipo inteiro, correspondente a essa posição.

Em C# existem essencialmente três tipos de arrays: *uni-dimensionais*, *bi-dimensionais* (também chamados *matrizes* ou *tabelas*) ou *jagged arrays* (arrays de arrays).

A class *Arrays* a seguir apresentada, mostra algumas aplicações destes tipos de arrays:

```

public void Arrays()
{
    //arrays unidimensionais
    int[] valores = { 1, 2, 3, 4, 5, 7 };
    string[] nomes = new string[2];
    nomes[0] = "Benfica";
    nomes[1] = "Mais Benfica";
    //mostrar array
    foreach (string s in nomes) Console.WriteLine(s);

    //arrays bidimensionais: Matrizes
    int[,] mat = new int[2, 3];
    mat[0, 1] = 12;
    //array de strings
    string[,] notas = new string[,];
    { { "Benfica", "+Benfica", "++Benfica" }, { "Porto", "-Porto", "--Porto" } };
    notas[0, 0] = "ok";
    //Número de colunas do array
    Console.WriteLine(notas.GetLength(1));

    //Jagged Arrays: arrays de arrays
    int[][] vals = new int[2][];
    vals[0] = new int[2];
    vals[1] = new int[] { 1, 2, 3, 4, 5 };
    vals[0][1] = 12;
    //Dimensão da primeira linha do array
    Console.WriteLine(vals[0].Length);
}

```

Uma utilização possível desta classe pode ser obtida através da utilização do construtor de class Arrays(), tal como a seguinte se representa:

```

namespace sobreArrays
{
    /// <summary>
    /// 1 - Revisões sobre arrays
    /// by lufer
    /// </summary>
    /// <remarks>
    /// Arrays simples
    /// Arrays multidimensionais
    /// Jagged Arrays
    /// </remarks>
    class Program
    {
        static void Main(string[] args)
        {
            new Program().Arrays();
        }
    }
}

```

## Operações avançadas sobre Arrays

O array é uma das estruturas mais utilizadas em programação, ao ponto de o próprio C# ter um NameSpace disponível que suporta o seu comportamento. Isto será abordado aquando o tratamento de *Colletions*, numa série posterior deste documento.

Das operações mais comuns sobre arrays ressaltam as operações de Procura e Ordenação.

### Ordenação

Ordenar um array envolve um conjunto de operações mais ou menos complexas. Existe um conjunto significativo de algoritmos de ordenação de arrays, uns mais complexos e mais eficientes e outros menos. Neste documento vamos apresentar a implementação em C# do algoritmo de ordenação de arrays chamado *Bubble Sort* (Figura 1).

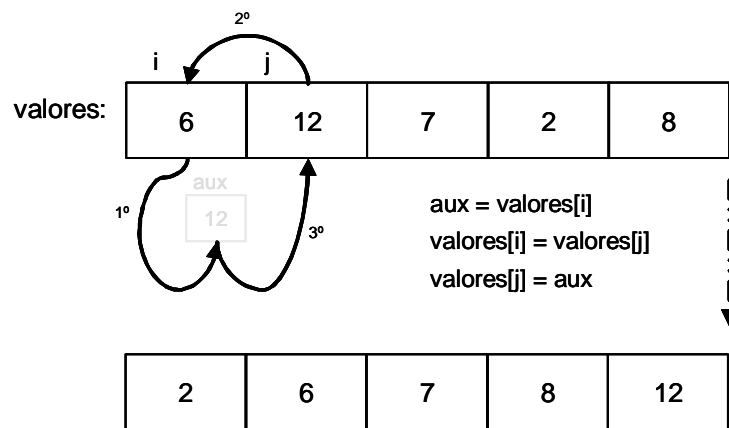


Figura 11 – Bubble Sort

O algoritmo implementado poderia ficar<sup>3</sup>:

```

/// <summary>
/// Ordenar um array de inteiros
/// </summary>
/// <param name="valores">array a ordenar</param>
public void OrdenaArray(int[] valores)
{
    int i, j;

    for(i=0; i<valores.Length-1; i++)
        for (j=i+1; j<valores.Length; j++)
            if (valores[i] > valores[j])           //troca
            {
                int aux = valores[i];
                valores[i] = valores[j];
                valores[j] = aux;
            }
}

```

Uma possível invocação deste método poderia ser:

```

static void Main(string[] args)
{
    GereArrays g = new GereArrays();
    g.OrdenaArray(new int[] { 2, 4, 3, 2 });
}

```

<sup>3</sup> Considere-se uma classe *GereArrays* na qual todos os métodos apresentados estão definidos

A ordenação de arrays deve contudo abarcar âmbitos bem mais complexos, nomeadamente quando o conteúdo do array forem objectos. Este assunto será tratado quando foram dadas as *Colletions* e os interfaces *IComparable*, à frente nesta sebenta.

## Procura

Sempre que estas estruturas de dados são utilizadas é muito frequente a necessidade de encontrar um elemento em particular. Tratam-se de algoritmos de pesquisa ou procura. De seguida vamos implementar um método (*Existe*) que se encarrega de percorrer todo o array na tentativa de encontrar um dado valor. Caso encontre deverá devolver o booleano *true*.

```
/// <summary>
/// Verifica se determinado valor existe num array
/// </summary>
/// <param name="valores">array</param>
/// <param name="v">valor a procurar</param>
/// <returns>true ou false</returns>
public bool Existe(int[] valores, int v)
{
    foreach (int x in valores)
    {
        if (x == v) return true;
    }
    return false;
}
```

Uma possível utilização deste método poderia ser:

```
GereArrays g = new GereArrays();
Console.WriteLine("Existe =" + g.Existe(new int[] { 2, 4, 3, 2 }, 8));
```

Cujo resultado seria *"Existe=False"*

Outra forma de implementar o mesmo método poderia ser:

```
public bool Existe(int[] valores, int v)
{
    for(int i=0;i<valores.Length;i++)
    {
        if (valores[i] == v) return true;
    }
    return false;
}
```

Agora uma possível utilização deste método poderia ser

```
int[] valores = new int[] { 2,4,3,1};
Console.WriteLine("Existe =" + g.Existe(valores, 4));
```

Que neste caso devolveria “Existe=True”.

### Passagem de Arrays como parâmetros

Os exemplos anteriores mostram também a forma de utilizar um array como parâmetro de um método. Repare-se nos argumentos do método *Existe* ou *OrdenaArray* e na forma como foram evocados.

Uma sintaxe possível para a utilização de arrays como parâmetros num método, poderá ser definida como:

*Definição do método:*

```
public/private [static] [tipo_dados_retorno|void] NomeMetodo (Tipo_Dados_Array[] nomeArray)
```

Exemplo:

```
public bool Existe(int[] valores, int v)
```

*Invocação do método*

```
objecto.NomeMetodo(nomeArray)
```

Exemplo:

```
int[] valores = new int[] { 2,4,3,1};
Console.WriteLine("Existe =" + g.Existe(valores, 4));
```

Na passagem de um array (com o nome do array) como um parâmetro de um método, o que é de facto passado é o endereço de memória da primeira posição do array. As alterações ao parâmetro array efectuadas no interior do método são efectuadas sobre o array real.

### Métodos com número variável de parâmetros (*params*)

O número de parâmetros a passar para um método poderá não ser sempre bem definido. Poderá acontecer que em contextos diferentes existam mais ou menos parâmetros que interesse tratar. Isto é implementado através de um *array de parâmetros* (*params*) de qualquer tipo (veremos mais à frente o significado real de objecto).

Analisemos o seguinte exemplo:

```
public void Write(string s1, string s2)
{
    Console.WriteLine("S1={0} e S2={1}", s1, s2);
}

public void Write(string s, params string[] strs){
    Console.WriteLine(s);
    for (int i = 0; i < strs.GetLength(0); i++)
    {
        Console.WriteLine(strs[i]);
    }
}
```

Como se pode analisar, deparamo-nos com um *overloading*<sup>4</sup> do método *Write* (o mesmo método surge definido de duas formas diferentes). No primeiro caso estão previstos dois argumentos. No segundo caso, um dos argumentos é um array de strings classificado por *params*.

Caso seja evocado o método *Write("Viva","Benfica")*, será a primeira definição do método *Write* a ser executada.

Caso o número de parâmetros na invocação seja superior a dois, então será executada a segunda definição do método. É o caso de *Write("Viva","o","Benfica")*.

O excerto de código seguinte mostra a aplicação deste tipo de métodos.

```
g.Write("Viva o", "Benfica");

//Resultado: Viva o Benfica"

string[] strs = new string[3];
strs[0] = "Viva";
strs[1] = "o";
strs[2] = "Benfica";

g.Write("Viva o Benfica", strs);

//Resultado:
//Viva
//o
//Benfica
```

---

<sup>4</sup> *Overloading* será analisado aquando o tratamento de classes

```
g.Write("Viva", "o", "Benfica");

//Resultado:
//Viva
//o
//Benfica
```

O caso apresentado refere a utilização de arrays de *strings*. Mas pode ser aplicado a qualquer tipo de dados. Este tipo de métodos com a aplicação de *params arrays* serão generalizados mais à frente aquando o tratamento de Objectos (Parte IV desta Sebenta).

Um outro caso que evidencia a importância deste tipo de métodos pode ser dado pelo exemplo seguinte. Calcular o somatório dos valores de um array.

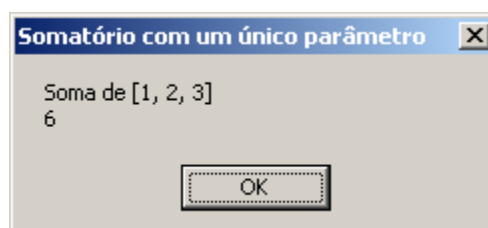
Na forma normal de tratamento de um array de inteiros, a soma poderá ser implementada com o seguinte algoritmo:

```
/// <summary>
/// Somatório de um array de inteiros.
/// Um só parametro
/// </summary>
/// <param name="v">array de inteiros</param>
/// <returns>a soma total</returns>
public int Somatorio1(int[] v)
{
    int s = 0;
    for (int i = 0; i < v.Length; i++)
    {
        s += v[i]; //equivalente a s = s+v[i];
    }
    return s;
}
```

E uma invocação possível deste método (com um só parâmetro) poderia ser :

```
int[] x = new int[] { 1,2,3};
MessageBox.Show("Soma de [1, 2, 3] \n"+ga.Somatorio1(x).ToString(),"Somatório com um único parâmetro");
```

Cujo resultado seria:





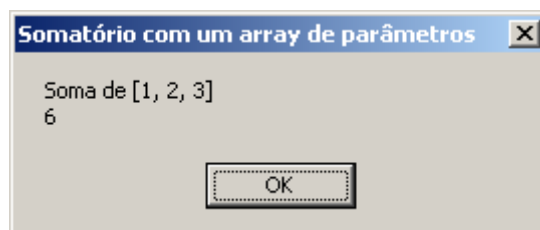
Utilizando um array de parâmetros, ficaria:

```
/// <summary>
/// Somatório de um array de inteiros.
/// Com numero de params variavel
/// </summary>
/// <param name="v"></param>
/// <returns></returns>
public int Somatorio(params int[] v) //diferente de (int[] v)
{
    int s = 0;
    for (int i = 0; i < v.Length; i++)
    {
        //s = s + v[i];
        s += v[i];
    }
    return s;
}
```

E uma invocação possível poderia ser:

```
MessageBox.Show("Soma de [1, 2, 3] \n" + ga.Somatorio(1, 2, 3).ToString(), "Somatório com um array de parâmetros");
```

E cujo resultado (igual) seria:



O exemplo seguinte mostra um método que apresenta o conteúdo de um array de qualquer tipo (generalizado com o tipo *object*).

```

/// <summary>
/// Métodos com argumentos dinâmicos (params array)
/// </summary>
/// <param name="s1">string</param>
/// <param name="s2">arrays de strings</param>
public string Show(params object[] args)
{
    string aux = "-";
    for (int i = 0; i < args.Length; i++)
    {
        aux += args[i].ToString() + "-";
    }
    return aux;
}

```

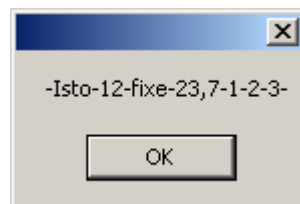
Este método pode ser evocado por:

```

GereArrays ga = new GereArrays();
MessageBox.Show(ga.Show("Isto", 12, "fixe", 23.7, 1, 2, 3));

```

E o resultado seria,



### Comparando “params” com “...” em JAVA

A versão *JDK5* já implementa de forma semelhante a lista de argumentos de tamanho variável (*Variable-Length Argument Lists*). O operador usado é a *elipsis* (...). Um tipo de argumento seguido deste operador indica que o método recebe um número variável de argumentos desse tipo. O exemplo seguinte mostra como se pode usar no cálculo da média de um conjunto de valores.

```

//Adaptado de www.deitel.com
//by lufer
public class VarargsTest
{
    // calcula a media
    public static double average( double... numbers )
    {
        double total = 0.0;
        for ( double d : numbers )
            total += d;
        return total / numbers.length;
    }

    public static void main( String args[] )
    {
        double d1 = 10.0;
        double d2 = 20.0;
        double d3 = 30.0;
        double d4 = 40.0;

        System.out.printf( "d1 = %.1f\nd2 = %.1f\nd3 = %.1f\nd4 = %.1f\n\n",
            d1, d2, d3, d4 );

        System.out.printf( "Média de d1 e d2 é: %.1f\n",
            average( d1, d2 ) );
        System.out.printf( "Média de d1, d2 e d3 é: %.1f\n",
            average( d1, d2, d3 ) );
        System.out.printf( "Média de d1, d2, d3 e d4 é: %.1f\n",
            average( d1, d2, d3, d4 ) );
    }
}

```

O resultado obtido seria:

```

d1 = 10,0
d2 = 20,0
d3 = 30,0
d4 = 40,0

Media de d1 e d2 e: 15,0
Media de d1, d2 e d3 e: 20,0
Media de d1, d2, d3 e d4 e: 25,0

```

## Exemplo completo

```
using System;
using System.Text;

namespace sobreArrays
{
    /// <summary>
    /// 1 - Operações avançadas sobre Arrays
    /// by lufer
    /// </summary>
    class GereArrays
    {
        /// <summary>
        /// Ordenar um array de inteiros
        /// </summary>
        /// <param name="valores">array a ordenar</param>
        public void ordenaArray(int[] valores)
        {
            int i, j;

            for(i=0;i<valores.Length-1;i++)
                for (j=i+1;j<valores.Length;j++)
                    if (valores[i] > valores[j]) //troca
                    {
                        int aux = valores[i];
                        valores[i] = valores[j];
                        valores[j] = aux;
                    }
        }
    }
}
```

```

    /// <summary>
    /// Verifica se determinado valor existe num array
    /// </summary>
    /// <param name="valores">array</param>
    /// <param name="v">valor a precurar</param>
    /// <returns>true ou false</returns>
    /// <<< remarks>Implementado com foreach()</remarks>
    public bool Existe1(int[] valores, int v)
    {
        foreach (int x in valores)
        {
            if (x == v) return true;
        }
        return false;
    }

    /// <summary>
    /// Verifica se determinado valor existe num array
    /// </summary>
    /// <param name="valores">array</param>
    /// <param name="v">valor a precurar</param>
    /// <returns>true ou false</returns>
    /// <remarks>Implementado com ciclo for()</remarks>

    public bool Existe(int[] valores, int v)
    {
        for(int i=0;i<valores.Length;i++)
        {
            if (valores[i] == v) return true;
        }
        return false;
    }

    public void Write(string s1, string s2)
    {
        Console.WriteLine("S1={0} e S2={1}", s1, s2);
    }

    /// <summary>
    /// Métodos com argumentos dinâmicos (params array)
    /// </summary>
    /// <param name="s1">strings</param>
    /// <param name="s2">arrays de strings</param>
    public void Write(string s, params string[] strs){
        Console.WriteLine(s);
        for (int i = 0; i < strs.GetLength(0); i++)
        {
            Console.WriteLine(strs[i]);
        }
    }

```

```

//Teste
static void Main(string[] args)
{
    new GereArrays().Arrays();

    GereArrays g = new GereArrays();
    g.ordenaArray(new int[] { 2, 4, 3, 2 });

    int[] valores = new int[] { 2, 4, 3, 1 };
    Console.WriteLine("Existe =" + g.Existe(valores, 4));

    g.Write("Viva o", "Benfica");

    //Resultado: Viva o Benfica

    string[] strs = new string[3];
    strs[0] = "Viva";
    strs[1] = "o";
    strs[2] = "Benfica";

    g.Write("Viva o Benfica", strs);

    //Resultado:
    //Viva
    //o
    //Benfica

    g.Write("Viva", "o", "Benfica");

    //Resultado:
    //Viva
    //o
    //Benfica
}

```

## Exercícios

Procure implementar os seguintes métodos:

- Ordenar um array de forma crescente
- Calcular o maior e menor elementos de um array
- Copiar um array para outro
- Implementar um método que consegue ordenar uma array de qualquer tipo.

## Referências

<http://www.softsteel.co.uk/tutorials/cSharp/>

<http://www.csharp-station.com/Tutorial.aspx>

<http://csharpcomputing.com/Tutorials/TOC.htm>

<http://www.softsteel.co.uk/tutorials/cSharp/lesson13.html>

C# School – Programmers Heaven

## Parte IV – Recursividade

Autores: Célio Carvalho; Luís Ferreira

---

Esta quarta parte deste documento C# Essencial, dá início ao estudo da Recursividade. É da responsabilidade do aluno Célio Carvalho, Nº 6474

### Recursividade:

Para muitos sinónimo de “Stack Overflow” ou “error: Stack Fault”, e para outros uma forma de partir um problema em vários de menor dimensão, o paradigma da “recursividade” é “tabu” para uns e uma forma de simplificar para outros.

No final deste capítulo, o aluno terá a capacidade de identificar um bloco de código que faz uso da recursividade, e será também capaz de desenvolver correctamente um método recursivo em C#, aproveitando as vantagens e atendendo às desvantagens do seu uso.

### Definição:

Antes de mais interessa dizer que um método recursivo é aquele que se “chama” (invoca) a si próprio.

```
... Soma (...)  
    {  
        ...;  
        Soma (...);  
        ...;  
    }
```

O uso da recursividade advém de uma estratégia do programador dividir um problema em “partes mais pequenas”, tornando-o assim de mais fácil compreensão e resolução.

Todos nós executamos “tarefas diárias” de forma recursiva apesar de não termos consciência disso. Veremos alguns exemplos onde usamos a recursividade no nosso dia a dia.

### Exemplos:

#### ***EXEMPLO 1: Lavar a louça do jantar***



Podíamos escrever rapidamente o algoritmo de lavar manualmente a louça do jantar. No entanto todos o imaginam grosso modo pelo que não o faremos aqui.

Imaginamos por certo que existirá algures lá no meio, um ciclo onde serão executadas as tarefas a serem repetidas para cada prato, como por exemplo: pegar no prato, passar por água, esfregar, enxaguar, etc.

A questão é: *qual o tipo de ciclo mais adequado a usar neste caso?*

Se fizermos um “brainstorming” acerca de soluções para este algoritmo, aparecerão com toda a certeza, varias sugestões usando diferentes tipos de ciclos.

Existirá, no entanto, uma questão comum que será tida em linha de conta entre as diversas abordagens: Quando deverá terminar o ciclo? (como controlar o fim do ciclo). Existirão por certo sugestões: contar previamente os pratos a lavar; haver uma verificação da existência de algum prato na mão no início de cada pulo, etc.

A verdade é que nenhuma dessas sugestões parece teatralizar “fielmente” a condição de paragem desta tarefa que é executada tão naturalmente pelo ser humano. Não é expectável que alguém pegue num prato e valide de seguida se tem algum prato na mão para lavar, nem tão pouco existirá alguém que conte os pratos previamente, para saber quando terminar o ciclo de lavagem dos pratos.

Se analisarmos bem esta questão, o que uma pessoa faz quando lava os pratos do jantar, é uma tarefa repetitiva sim, mas não é um ciclo na verdadeira acepção da palavra. Na realidade, usa algo de muito mais fácil compreensão. Vai “pegando” e “lavando” cada prato, até que chega ao último e pára de lavar pratos porque já os lavou todos.

No fundo, vai lavando repetidamente cada prato, até que todos estejam lavados. Quando todos os pratos estiverem lavados, termina a tarefa de lavar pratos e começa outra qualquer.

O pseudocódigo para retratar este exemplo, poderia ser algo do género:

```
... LavaPrato(...)
{
    Se (todosPratosLavados==true)
    {
        Return          // Os pratos já estão todos
                        // lavados, então terminar
    }

    ... // pegar
    ... // passar por água
    ... // esfregar
    ... // enxaguar

    LavaPrato(...)
        // Lava o próximo prato, invocando-se a si
        // próprio de novo.
        // Cria uma nova instância de si próprio.
```

```
}
```

Para terminar este exemplo, dêmos alguma ênfase à instrução que valida (antes de qualquer outra) a existência ou não de mais pratos para lavar (`todosPratosLavados==true`). Veremos ao longo deste capítulo, que a condição de paragem na recursividade assume um papel importantíssimo que poderá ditar um bom ou mau uso de métodos recursivos em programação.

### EXEMPLO 2: Caminhar

Outro exemplo (agora apresentado de forma mais sintética) seria a tarefa humana de “caminhar”.

Caminhamos para chegar aos destinos pretendidos e caminhar não é mais nem menos que dar um passo após o outro repetidamente, até chegar ao destino.

Fica uma sugestão em pseudocódigo do que seria dar um passo para percorrer determinado trajecto:

```
... DaPasso (...)
{
    Se (destinoAtingido==true)
    {
        Return
        // Já chegamos onde queríamos, não
        // queremos dar mais passos,
        // portanto termina.
    }

    DaPasso (...)
    // Dá um novo passo, invocando-se
    // a si próprio de novo.
    // Cria uma nova instância de si próprio.
}
```

### EXEMPLO 3: Factorial

Neste exemplo pretende-se demonstrar o potencial da recursividade em programação e muito especificamente em C#.

Antes de mais, lembrar que calcular o factorial de um número, não é mais que multiplicar a acumulação dos sucessivos produtos pelo número anterior, até que se chegue a 1. Por exemplo, o factorial de 5 é  $((5 \times 4) \times 3) \times 2 \times 1 = 120$ .

A redução a código desta multiplicação poderia ser implementada pelo seguinte método que usa um ciclo iterativo para calcular o factorial de qualquer número.

```
(...)
```

```
public static int FactorialIterativo(int x)
{
    int resultado = 1;

    for (int i = 2; i <= x; i++)
```

```

        {
            resultado *= i;
        }

        return resultado;
    }
    (...)

```

No entanto a definição de factorial de um número também pode ser descrita de outra forma. Podemos dizer que o factorial de um dado número  $x$ , é o resultado da multiplicação desse número pelo factorial de  $x - 1$ .

Descrevendo de forma mais visual, teríamos que o factorial de 5 seria calculado da seguinte forma:

$5! = 5 * 4!$

Vemos, no entanto, que para calcular o factorial de 5, teremos que conhecer primeiro o factorial de 4 e assim será sucessivamente até chegarmos ao factorial de 1 que é simplesmente 1:

$4! = 4 * 3!$   
 $3! = 3 * 2!$   
 $2! = 2 * 1!$   
 $1! = 1$

Com esta definição, é perfeitamente perceptível a dependência do cálculo factorial de um número, do mesmo cálculo aplicado primeiramente ao número antecessor. É, portanto um caso típico em que o uso de um método recursivo, como aquele mostrado a seguir, parece a solução natural para este problema.

```

(...)
public static int FactorialRecursivo(int x)
{
    int valorDevolver, valorDevolvido;

    if (x == 0)
    {
        valorDevolver = 1;
    }
    else
    {
        valorDevolvido = FactorialRecursivo(x - 1);
        valorDevolver = x * valorDevolvido;
    }

    return valorDevolver;
}
(...)

```

Se pretendêssemos calcular com este método o factorial de 5, chamaríamos o método como faríamos com qualquer outro (ex: Factorial(5);).

A sequência recursiva de chamadas deste método iria acontecer pela ordem definida a seguir:

1º - cria instância 1                       $5! = 5 * 4!$  *aguarda*

---

2º - cria instância 2	4!	=	4 * 3!	<i>aguarda</i>
3º - cria instância 3	3!	=	3 * 2!	<i>aguarda</i>
4º - cria instância 4	2!	=	2 * 1!	<i>aguarda</i>
5º - cria e termina instância 5	1!	=	1	<i>paragem</i>
6º - termina instância 4	2!	=	2	<i>retoma</i>
7º - termina instância 3	3!	=	6	<i>retoma</i>
8º - termina instância 2	4!	=	24	<i>retoma</i>
9º - termina instância 1	5!	=	120	resultado final

*aguarda* - antes de fazer esta multiplicação, o sistema precisa de um valor que ainda não conhece (factorial de outro numero), portanto coloca esta instância em stand by até que seja apurado o valor em falta, e acrescenta-a à pilha (empilhamento).

*paragem* - ao atingir a condição de paragem devolve o valor solicitado e termina-se.

*retoma* - o valor que precisava para calcular o produto já lhe foi comunicado logo já pode concluir a multiplicação, devolver o valor do factorial que lhe fora solicitado e terminar-se eliminando-se de seguida da pilha (desempilhamento).

## Síntese

Um método recursivo é então aquele que se invoca directamente a si próprio, ou que invoca um segundo que por sua vez chama de novo o primeiro de forma indirecta.

Invocar um método recursivo é igual à invocação de qualquer outro método. No entanto, no método recursivo, o sistema tem a necessidade de guardar a informação inerente a cada instância até que essa mesma termine. Para isso, sempre que é criada uma nova instância de um método, o seu estado e o valor dos seus argumentos, vão sendo guardados numa pilha (stack) onde a gestão de entradas e saídas é baseada no critério LIFO (ultimo a entrar é o primeiro a sair).

Por fim salientar que, o estado e os parâmetros passados para cada instância, são visíveis somente dentro daquela a que diz respeito.

## Características

As características principais de um método recursivo são:

- Ponto de paragem que é no fundo o limite preestabelecido pelo programador, até o qual o método se vai “auto invocando”. Nos exemplos vistos anteriormente, os pontos de paragem usados foram:

- (todosPratosLavados==true) – Quando a condição for verdadeira (significa que todos os pratos já estão lavados) termina a execução da instancia actual LavaPrato(...) retirando-a da pilha, permitindo que a instância anterior (aquela que fica no topo depois desta ter sido retirada), possa continuar a sua execução e também termine.
  - (destinoAtingido==true) – Quando a condição for verdadeira (significando que já se chegou ao destino), termina a execução da instância actual DaPasso(...), retirando-a da pilha, permitindo assim que a instância anterior (agora ultima da pilha) também se termine e assim sucessivamente até terminar a primeira de todas elas;
  - (x==0) – Quando a condição for verdadeira (ou seja quando x chegar a zero), termina a execução da instância actual FactorialRecursivo(int x) comunicando o valor 1 à instancia anterior, para que esta continue o seu trabalho. De seguida o sistema retira-a da pilha pelo facto da instancia em questão já se ter extinguido.
- Divisibilidade – é a possibilidade oferecida pela recursividade que permite resolver um problema através da invocação recursiva de casos mais pequenos, sendo estes resolvidos através da resolução de outros ainda mais pequenos e assim sucessivamente até atingir o ponto de paragem que inicia a finalização em lote de todas instâncias aplicando o critério LIFO à pilha.
  - Combinação – traduz a capacidade da instância activa (ultima da pilha) poder devolver informação à que a antecede (para que esta a manipule também e devolva à sua anterior), antes de ser retirada da pilha, dando assim o seu contributo para a resolução do problema original.

## Recursividade versus Ciclos

Mesmo para os amantes da recursividade, nem sempre a implementação de ciclos através de métodos recursivos vem a traduzir-se na melhor opção para um dado contexto.

Na fase de desenvolvimento, os testes de carga poderão ser uma ferramenta útil para contrapor ambas as abordagens (recursividade ou ciclos iterativos) com vista a escolha do caminho mais eficiente para a resolução do problema.

## Vantagens e Desvantagens

Os métodos recursivos produzem um código de mais fácil leitura e compreensão.

Por outro lado, por usarem intensivamente a pilha (precisando alocar e desalocar recursos constantemente), tendem a ser mais lentos quando comparados com ciclos iterativos. No entanto, por vezes pode valer a pena sacrificar a eficiência em detrimento da facilidade de leitura e compreensão do código.

De realçar por fim, uma última desvantagem relacionada com a maior dificuldade na depuração nos casos em que se opte pela via da recursividade.

## Conclusão

Na conclusão deste capítulo, faz sentido lembrar que um método recursivo é aquele que faz pelo menos uma invocação a si próprio seja directamente ou indirectamente (por via de um segundo método que volta a invocar o primeiro). Faz sentido também reter que um método recursivo tem que ter obrigatoriamente uma condição de paragem bem definida para que não seja invocado infinitamente e produza efeitos perversos.

Por fim, a recursividade deve ser usada quando é a forma recursiva a mais simples e intuitiva de programar numa solução para um dado problema.

## Referências

- <http://pt.wikipedia.org/wiki/Recursividade>
- [http://intranet.deei.fct.ualg.pt/PI\\_flobo/teorica12.html](http://intranet.deei.fct.ualg.pt/PI_flobo/teorica12.html)
- <http://www.estig.ipbeja.pt/~rmcp/estig/2001/1s/lp1/teorica/recursividade.pdf>

## Parte V – Programação Orientada a Objectos

Autores: João Carlos e Luís Ferreira

---

Esta quarta parte deste documento C# Essencial, dá início ao estudo da programação orientada por objectos.

### Conceito de classe

Uma classe é um modelo abstracto utilizado para definir novos tipos de dados. Uma classe pode conter estruturas de dados, operações e propriedades.

Define-se uma classe através da palavra reservada *class*:

```
class MyClass
{
    // campos, métodos e propriedades
}
```

*MyClass* é o nome da classe.

### Objectos

Um objecto é uma instância obtida a partir do modelo especificado por uma classe. Para criar um objecto, utiliza-se a palavra reservada *new*:

```
MyClass myObjectReference = new MyClass();
```

No exemplo anterior, criamos um objecto do tipo *MyClass*. O objecto encontra-se referenciado pelo identificador *myObjectReference*.

Exemplo completo de uma Classe:

```

class Student1
{ // campos da classe Student
    string name;
    int age;
    int marksInMaths;
    int marksInEnglish;
    int marksInScience;
    int totalMarks = 60; // inicialização
    int obtainedMarks;
    double percentage;
}

```

A classe *Student1* define vários dados relacionados com um estudante (name, age, etc). As variáveis podem ser inicializadas aquando da sua definição (ex: *totalMarks*).

## Métodos

Os métodos são as operações que podem ser executadas sobre os dados. Um método pode receber dados de entrada através dos seus parâmetros e pode devolver um resultado de um determinado tipo de dados:

```

<return type> <name of method>(<data type> <identifier>, <data type> <identifier>,...)
{
    // body of the method
}

```

Por exemplo o método *FindSum* recebe dois inteiros por parâmetro e devolve como resultado um valor do tipo inteiro. O resultado do método é devolvido via comando *return*. No caso dos métodos sem resultado deve ser utilizado a palavra *void* na especificação do tipo do resultado.

```

int FindSum(int num1, int num2)
{
    int sum = num1 + num2;
    return sum;
}

void ShowCurrentTime()
{
    Console.WriteLine("A hora actual é: " + DateTime.Now);
}

```

O método *ShowCurrentTime* não recebe nenhum parâmetro e não devolve nenhum resultado pelo que o tipo aplicado no resultado é *void*. O método *ShowCurrentTime* quando executado escreve na consola a data e a hora actual.



## Instâncias

Na linguagem C# uma classe é instanciada (criação de um objecto) aplicando o comando *new*. O comando *new* permite criar um objecto de uma dada classe, devolvendo como resultado uma referência para o objecto criado. Por exemplo a instrução:

```
Student theStudent = new Student();
```

permite criar uma instância (objecto) da classe *Student*. A referência do objecto encontra-se armazenada na variável *theStudent*.

## Modificadores de acesso

A linguagem de programação C#, disponibiliza modificadores de acesso para restringir o acesso a determinado variável ou método. Existem 5 modificadores de acesso que podem ser aplicados a qualquer membro de uma classe. Por ordem decrescente de restrição temos:

1. *private*, acesso exclusivamente a partir da mesma classe;
2. *protected internal*;
3. *internal*;
4. *protected*;
5. *public*, sem restrições de acesso;

A classe *Student2* apresentada a seguir define três variáveis associando-as ao modificador de acesso *private* e 4 métodos associando-os ao modificador de acesso *public*. As variáveis podem ser acedidas somente a partir da classe *Student2* (*private*) enquanto que os métodos não estão associados a nenhuma restrição de acesso (*public*).

```

class Student2
{
    // campos
    private int number;
    private double mark1;
    private double mark2;

    // métodos
    public void setNumber(int n)
    {
        number = n;
    }
    public void setMark1(double v)
    {
        mark1 = v;
    }
    public void setMark2(double v)
    {
        mark2 = v;
    }
    public double CalculateFinalMark()
    {
        double res = (mark1 + mark2) / 2;
        return res;
    }
}

```

Na classe *Test* (exemplo seguinte), a instrução `Student2 theStudent = new Student2();` permite criar uma instância (objecto) da classe *Student2*. A referência do objecto encontra-se armazenada na variável *theStudent*. Dada a definição da classe *Student2*, cada uma das suas instâncias contém as variáveis *number*, *mark1* e *mark2*. Por outro lado é possível executar os métodos *setNumber*, *setMark1*, *setMark2* e *CalculateFinalMark* sobre qualquer instância da classe *Student2*.

```

class Test
{
    public static void Main()
    {
        Student2 theStudent = new Student2();
        theStudent.setNumber(1234);
        theStudent.setMark1(15.6);
        theStudent.setMark2(16.3);
        Console.WriteLine(theStudent.CalculateFinalMark());
    }
}

```

O código da classe *Test* permite então criar um objecto do tipo *Student2*. A atribuição de valores às variáveis *number*, *mark1* e *mark2* do objecto *theStudent* são realizadas, invocando os métodos públicos *setNumber*, *setMark1* e *setMark2* respectivamente. A expressão *theStudent.CalculateFinalMark()* permite por sua vez

executar o método *CalculateFinalMark()* sobre os dados do objecto referenciado pela variável *theStudent*. Isto é, permite calcular a nota final do aluno.

## Membros *static* de uma classe

As variáveis e os métodos definidos até agora são membros de instância de uma determinada classe. Por outras palavras estes membros fazem sempre parte dos objectos criados. Por exemplo todos os objectos criados a partir da classe *Student2* contêm as suas próprias variáveis e métodos (*number*, *mark1*, *mark2*, etc).

Os membros *static* de uma classe (variáveis/métodos de classe) podem ser partilhados com todos os objectos da classe. Vejamos uma nova definição da nossa classe *Student2*.

```
class Student2
{
    // campos
    private static string institution="IPCA";
    private int number;
    private double mark1;
    private double mark2;

    // métodos
    public void setNumber (int n)
    {
        number = n;
    }
    public void setMark1 (double v)
    {
        mark1 = v;
    }
    public void setMark2 (double v)
    {
        mark2 = v;
    }
    public double CalculateFinalMark()
    {
        double res = (mark1 + mark2)/2;
        return res;
    }
}
```

Agora nesta definição de *Student2*, a expressão *private static string institution="IPCA";* define uma variável de classe, a qual é partilhada com todos os objectos. O acesso aos membros *static* realiza-se a partir do nome da classe. Exemplo: *Student2.institution*

## Construtores

Um construtor é um método especial.

1. Tem o nome da classe
2. Não devolve nenhum resultado
3. Contem código para a inicialização de objectos
4. É invocado aquando da criação de um objecto

Continuando com a classe *Student2*, vejamos como definir um seu construtor<sup>5</sup>:

```
class Student2
{
    // campos
    private static string institution="IPCA";
    private int number;
    private double mark1;
    private double mark2;

    // constructor
    public Student2(int n, double m1, double m2)
    {
        number = n;
        mark1 = m1;
        mark2 = m2;
    }

    // métodos
    public void setNumber(int n)
    {
        number = n;
    }
    public void setMark1(double v) [...]
    public void setMark2(double v) [...]
    public double CalculateFinalMark() [...]
}
```

A instrução seguinte mostra a aplicação desse construtor. Esta instrução permite então criar objectos da classe *Student2* inicializando-os logo à partida com o número do aluno e as respectivas classificações.

```
Student2 st = new Student2(1234, 12.5, 17.4);
```

---

<sup>5</sup> Note-se que as implementações de alguns dos métodos estão ocultas, na tentativa de minimizar espaço neste documento. Contudo a sua implementação está definido nas definições anteriores desta classe

## Propriedades

O C# apresenta um conceito interessante de manipulação de variáveis de instância (geralmente privadas ao exterior). Tratam-se das *Properties* (Propriedades) *set* e *get*.

Segundo os princípios do Paradigma Orientado aos Objectos, é de todo recomendável que as variáveis de instância seja encapsuladas de forma a não poderem ser manipuladas directamente do exterior. Isto consegue-se com o classificador *private*.

Neste sentido recomenda-se a criação de métodos *Set* e *Get* para a referida variável de instância. Por exemplo, na nossa classe *Student*, a variável *number* deverá possuir dois métodos, *SetNumber* e *GetNumber* capazes de a manipular. São lógicas operações do género:

```
Student2 st = new Student2();
st.SetNumber(1234);
Console.WriteLine("Número {0}", st.GetNumber());
```

Agora no C# esta tarefa fica bastante simplificada. As propriedades *set* e *get* vêm substituir a necessidade de definir este tipo de métodos. Vejamos como.

### Definição de propriedades *set* e *get*

Syntax:

```
public tipo nomePropriedade
{
    get { return(variavel);}
    set {variável=value;}
}
```

Na nossa classe *Student2*, a definição da propriedade para a variável *number* poderia ficar então:

```
/// <summary>
/// Property Number
/// </summary>
public int Number
{
    get { return (number); }
    set { if (value > 0) number = value; }
}
```

Dentro de cada bloco *get* e *set* é possível aplicar expressões C# normais de teste ou controlo.

## Utilização de Propriedades

Uma vez definidas, a sua utilização é idêntica tanto para a inserção como para a consulta de dados. A utilização é idêntica à forma de utilização de uma variável de instância pública. O nosso exemplo poderia ser reescrito da seguinte forma:

```
Student2 st = new Student2();
st.Number=1234;
Console.WriteLine("Número {0}",st.Number);
```

A nossa classe *Student2* poderia então ser reescrita da seguinte forma:

```
class Student2
{
    private static string institution="IPCA";
    private int number;
    private double mark1;
    private double mark2;
    //...
    /// <summary>
    /// Property Number
    /// </summary>
    public int Number
    {
        get { return (number); }
        set { if (value > 0) number = value; }
    }
    /// <summary>
    /// Property Mark1
    /// </summary>
    public double Mark1
    {
        get { return (mark1); }
        set { mark1 = value; }
    }
    /// <summary>
    /// Property Mark2
    /// </summary>
    public double Mark2
    {
        get { return (mark2); }
        set { mark2 = value; }
    }
    //...
```

Resumindo, as propriedades devem ser vistas como métodos particulares usados para aceder/actualizar os valores das variáveis do estado e não para calcular/processar os seus valores. Para isso existem os métodos normais.

### Override de Métodos

Em muitas circunstâncias a manipulação de objectos deveria ser facilitada. Por exemplo, deveria ser possível comparar directamente objectos (tipo `ObjectA` igual `ObjectB`), apresentar a informação do estado do objecto (tipo com a utilização do método `ToString()`), etc.

Este tipo de procedimentos é possível implementar com a definição de métodos *override*. Entenda-se *override* como a reescrita de métodos herdados das classes das quais se deriva. Isto é possível fazer essencialmente sobre os métodos `ToString()`, `Equal()`, `GetHashCode()` e os operadores `==` (igual) e `!=` (diferente). Ora vejamos como:

Rescrita do método `ToString()` (anote-se a palavra reservada *override*):

```
public override string ToString()
{
    return (String.Format("Aluno:{0}-({1})-({2})", number, mark1, mark2));
}
```

Reescrita do método `Equal()` (assume-se que dois `Students` são iguais se o *number* for igual):

```
public override bool Equals(Object obj)
{
    Student2 a = obj as Student2;
    return (this.number == a.number);
}
```

**Nota:** o operador *as* será analisado a seguir.

Reescrita do método `GetHashCode()` (importante no tratamento de colecções *HashTable*. A ver explorar mais à frente):

```
public override int GetHashCode()
{
    return (number);
}
```

Reescrita dos operadores `==` e `!=` (anote-se a palavra reservada *operator*):

```
//Reescrita dos Operadores == e !=
public static bool operator ==(Student2 x, Student2 y)
{
    return (x.Equals(y));
}

public static bool operator !=(Student2 x, Student2 y)
{
    return (!x.Equals(y));
}
```

Verifiquemos a importância do *override* de métodos através da análise dos seguintes exemplos:

Exemplo 1: ToString()

```
Student2 s1 = new Student2();
s1.Number = 1234;
s1.Mark1 = 10;

Console.WriteLine(s1.ToString());
```

O resultado obtido seria (procure verificar porquê?):

Aluno: 1234-(10)-(0)

Exemplo 2: Equal()

```
Student2 s1 = new Student2();
s1.Number = 1234;
s1.Mark1 = 10;

Console.WriteLine(s1.ToString());

Student2 s2 = new Student2();
s2.Number = 3001;
s2.Mark1 = 12;

Console.WriteLine(s1.Equals(s2));
```

O resultado obtido seria (procure verificar porquê?):

False



Exemplo 3: Operadores == e !=

```
bool y = s1 != s2;
Console.WriteLine(y);
```

O resultado obtido seria (procure verificar porquê?):

True

### Operadores *is* e *as*

Os operadores *is* e *as* permitem a generalização de métodos numa classe. Ou seja, o mesmo método consegue lidar com objectos de diferentes tipos. Analisemos os seguintes exemplos.

Assuma-se a existência das classes *Aluno* e *Curso*. O seguinte excerto mostra um método que consegue comparar tanto Alunos como Cursos. Anote a presença dos operadores *is* e *as*.

```
/// <summary>
/// Compara dois objectos
/// </summary>
/// <param name="x">Objecto</param>
/// <param name="y">Objecto</param>
/// <returns>true/false</returns>
public static bool Compara(Object x, Object y){

    //completar
    if (y == null) return true;

    if (x is Aluno && y is Aluno)
    {
        Aluno a = x as Aluno;
        Aluno b = y as Aluno;
        return(Aluno.comparaAlunos(a,b));
    }
    else
    {
        if (x is Clube && y is Clube)
        {
            Clube a = x as Clube;
            Clube b = y as Clube;
            return (a.NumSocios >= b.NumSocios);

        }
    }
    return false;
}
```

O exemplo seguinte mostra um método que consegue ordenar um array de objectos do tipo *Aluno* ou outros.

```

/// <summary>
/// Ordena um Array de Objectos
/// </summary>
/// <param name="arr">array a ordenar</param>
public static void Ordena(Object[] arr){
    if (arr[0] is Aluno)
    {
        for (int i = 0; i < arr.Length-1; i++)
        {
            for (int j = i + 1; j < arr.Length; j++)
            {
                if (Compara(arr[i],arr[j])){
                    Aluno t = new Aluno();
                    t = (Aluno)arr[j];
                    arr[j] = arr[i];
                    arr[i] = t;
                }
            }
        }
    }
}

```

### Exemplo completo

```

/// <summary>
/// Classe Aluno
/// </summary>
public class Aluno
{
    private string nome;
    private int idade;
    private string curso;
    private int numero;

    public Aluno(){
    }

    public Aluno(int idade, string nome, int numero)
    {
        this.idade = idade;
        this.nome = nome;
        this.numero = numero;
    }
}

```

```

//Propriedades
public int Numero
{
    get { return numero; }
    set
    {
        if (value < 0)
        {
            numero = 0;
        }
        else
            numero = value;
    }
}

public int Idade
{
    get { return idade; }
    set
    {
        if (value < 0)
        {
            idade = 0;
        }
        else
            idade = value;
    }
}

public string Nome
{
    get { return nome; }
    set { nome = value; }
}

public string Curso
{
    get{return curso;}
    set { curso = value; }
}

public static bool comparaAlunos(Aluno x, Aluno y)
{
    return (x.idade >= y.idade);
}

```

```

//-----
//Outros Métodos Úteis
//-----
/// <summary>
/// Completar uma Classe
/// </summary>
/// <returns></returns>
/// <remarks>
/// ToString()
/// Equals()
/// GetHashCode()
/// operator==
/// operator!=
/// </remarks>

public override string ToString()
{
    return (String.Format("{0}-({1})", nome, idade));
}

public override bool Equals(Object obj)
{
    Aluno a = obj as Aluno;
    return ((this.nome == a.Nome) && (this.idade == a.Idade));
}

public override int GetHashCode()
{
    return (numero);
}

//Reescrita dos Operadores == e !=
public static bool operator==(Aluno x, Aluno y)
{
    return (x.Equals(y));
}

public static bool operator!=(Aluno x, Aluno y)
{
    return (!x.Equals(y));
}
}

```

## Referências

<http://www.softsteel.co.uk/tutorials/cSharp/>

<http://www.csharp-station.com/Tutorial.aspx>

<http://csharpcomputing.com/Tutorials/TOC.htm>

[http://msdn2.microsoft.com/en-us/library/9b9dty7d\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/9b9dty7d(VS.80).aspx)

[http://msdn2.microsoft.com/en-us/library/2s05feca\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/2s05feca(VS.80).aspx)

C# School – Programmers Heaven

## Parte VI – Collections

Autores: João Carlos Silva e António Tavares

---

Esta parte do documento C# Essencial, dá início ao estudo das colecções na linguagem C# nomeadamente o estudo das classes *ArrayList*, *Stack*, *Queue*, *HashTable*, *SortedList* definidas no namespace *System.Collections* (*using System.Collections;*).

### ArrayList

A classe *ArrayList* é semelhante aos *arrays*. No entanto o seu tamanho é alterado dinamicamente à medida que os elementos são alterados. A instrução seguinte mostra como criar um objecto desta classe:

```
ArrayList list = new ArrayList();
```

O método *Add* permite adicionar elementos nos objectos da classe *ArrayList*:

```
list.Add(45);  
list.Add(87);  
list.Add(12);
```

O acesso aos dados do objecto pode ser efectuado como se de um array se tratasse (a instrução *Count* permite obter o número total de elementos presentes no objecto). O código seguinte implementa a listagem de todos os elementos de um *ArrayList*.

```
static void Main()  
{  
    ArrayList list = new ArrayList();  
    list.Add(45);  
    list.Add(87);  
    list.Add(12);  
    for (int i = 0; i < list.Count; i++)  
    {  
        Console.WriteLine(list[i]);  
    }  
}
```

**Métodos/propriedades definidos na classe *ArrayList* :**

- **Capacity** – devolve ou altera o número de elementos que a *ArrayList* pode conter.
- **Count** – número de elementos na *ArrayList*.
- **Add(object obj)** – adiciona o elemento obj na *ArrayList*.
- **Remove(object obj)** – remove o elemento obj da *ArrayList*.
- **RemoveAt(int i)** – remove um elemento da *ArrayList* na posição i
- **Insert(int i, object obj)** - insere o elemento obj na *ArrayList* na posição i.
- **Clear()** - Remove todos os elementos da *ArrayList*
- **Contains(object obj)** – Devolve um valor lógico consoante a *ArrayList* contém o elemento obj ou não.
- **IndexOf(object obj)** – Devolve o índice da primeira ocorrência de obj na *ArrayList*. Se não existir devolve -1

***Stack***

A classe *Stack* implementa o princípio *LIFO (Last In First Out)*. A instrução seguinte mostra como criar um objecto desta classe:

```
Stack st = new Stack();
```

**Métodos definidos na classe *Stack*:**

- **Push(object obj)** – adiciona o elemento obj no topo da pilha.
- **Pop()** – remove o elemento do topo da pilha.
- **Peek()** – devolve o elemento do topo da pilha.

```
static void Main()
{
    Stack stack = new Stack();
    stack.Push(2);
    stack.Push(4);
    stack.Push(6);
    Console.WriteLine("Número de elementos antes de Peek(): {0}", stack.Count);
    Console.WriteLine("Elemento Topo da Stack: {0}", stack.Peek());
    Console.WriteLine("Número de elementos após Peek(): {0}", stack.Count);
}
```

## Queue

A classe *Queue* implementa o princípio *FIFO* (*First In First Out*). A instrução seguinte mostra como criar um objecto desta classe:

```
Queue q = new Queue();
```

A classe *Queue* implementa os métodos *Enqueue* e *Dequeue* para, respectivamente, adicionar um novo elemento no fim da fila e retirar o primeiro elemento da fila.

```
static void Main()
{
    Queue queue = new Queue();
    queue.Enqueue(2);
    queue.Enqueue(4);
    queue.Enqueue(6);
    while (queue.Count != 0)
    {
        Console.WriteLine(queue.Dequeue());
    }
}
```

## HashTable

A classe *HashTable* permite manipular pares chave/valor. Cada valor de uma *HashTable* é identificado pela sua chave. Numa *HashTable* todas as chaves são únicas. O conceito é semelhante a um dicionário onde cada palavra (chave) está associada a sua definição (valor). A instrução seguinte mostra como criar um objecto desta classe:

```
HashTable ht = new HashTable();
```

O método *Add(object chave, object valor)* permite adicionar um novo elemento (chave/valor) numa *HashTable*:

```
ht.Add("st01", "Faraz");
ht.Add("sci01", "Newton");
ht.Add("sci02", "Einstein");
```

O método *Count* devolve o número de elementos (chave/valor) presentes na *HashTable*. O exemplo seguinte exemplifica o acesso ao valor de um par a partir da sua chave (*ht["st01"]*):

```
Console.WriteLine("Tamanho da Hashtable:{0}", ht.Count);
Console.WriteLine("Elemento com a chave: st01 é: {0}", ht["st01"]);
```



A remoção de um par a partir da sua chave é realizada através do método *Remove(Object obj)*. Vejamos o seguinte exemplo:

```
static void Main()
{
    Hashtable ht = new Hashtable(20);
    ht.Add("st01", "Faraz");
    ht.Add("sci01", "Newton");
    ht.Add("sci02", "Einstein");
    Console.WriteLine("Tamanho da Hashtable é: {0}", ht.Count);
    Console.WriteLine("Remover o elemento com a chave st01");
    ht.Remove("st01");
    Console.WriteLine("Tamanho da Hashtable após remoção: {0}", ht.Count);
}
```

As instruções *Keys* e *Values* permitem obter, respectivamente, o conjunto das chaves e o conjunto dos valores de uma *HashTable*:

```
static void Main()
{
    Hashtable ht = new Hashtable(20);
    ht.Add("st01", "Faraz");
    ht.Add("sci01", "Newton");
    ht.Add("sci02", "Einstein");
    Console.WriteLine("Mostrando chaves...");
    foreach (string key in ht.Keys)
    {
        Console.WriteLine(key);
    }
    Console.WriteLine("\nMostrando valores...");
    foreach (string Value in ht.Values)
    {
        Console.WriteLine(Value);
    }
}
```

O método *ContainsKey(Object obj)* permite determinar se uma chave pertence ou não à *HashTable*. Por sua vez o método *ContainsValue(Object obj)* permite determinar se uma chave pertence ou não à *HashTable*.

```
static void Main()
{
    Hashtable ht = new Hashtable(20);
    ht.Add("st01", "Faraz");
    ht.Add("sci01", "Newton");
    ht.Add("sci02", "Einstein");
    Console.WriteLine(ht.ContainsKey("sci01"));
    Console.WriteLine(ht.ContainsKey("st00"));
    Console.WriteLine(ht.ContainsValue("Einstein"));
}
```

Neste caso o resultado apresentado seria:

True

False

True.

## SortedList

A classe *SortedList* permite manipular pares chave/valor, ordenando a informação pela chave. Os dados podem ser acedidos a partir das chaves ou a partir dos índices dos pares. A instrução seguinte mostra como criar um objecto desta classe:

```
SortedList sl = new SortedList();
```

**Propriedades e métodos definidos na classe:**

- **Count** – número de elementos contidos na *SortedList*.
- **Keys** – devolve o conjunto das chaves da *SortedList*.
- **Values** – devolve o conjunto dos valores da *SortedList*.
- **Add(object key, object value)** – adiciona um novo par (chave, valor) na *SortedList*.
- **GetKey(int index)** – devolve a chave presente na posição *index*.
- **GetByIndex(int index)** – devolve o valor presente na posição *index*.
- **IndexOfKey(object key)** – devolve o índice de uma chave.
- **IndexOfValue(object value)** – devolve o índice de um valor.

- **Remove(object key)** – remove um par (chave/valor) a partir da sua chave.
- **RemoveAt(int)** – remove um par a partir do seu índice.
- **Clear()** - remove todos os pares da SortedList.
- **ContainsKey(object key)** – Devolve um valor lógico consoante a existência ou não da chave *key* no SortedList.
- **ContainsValue(object value)** - Devolve um valor lógico consoante a existência ou não do valor *value* no SortedList

### Exercício 1:

Resolva este exercício fazendo uso da classe *SortedList*

1. Pretende-se implementar uma agenda de contactos telefónicos.

Para representação de um contacto define a classe *Contacto* com duas variáveis de instância: *nome* e *telemóvel*;

Na classe *Contacto* define ainda os métodos/construtores seguintes:

```

Contacto()
Contacto(string nome, int telemóvel)

public string getNome(); //obter nome
public string getTelemóvel(); //obter telemóvel
public void setNome(string nome); //alterar nome
public void setTelefone(int telemóvel); //alterar telemóvel
public bool igual(Contacto c); //determinar igualdade entre dois contactos
public string toString(); //extrair dados de um contacto sob a forma de uma string

```

2. Define agora uma nova classe *Agenda* contendo uma identificação e uma lista de contactos  
Na agenda não podem existir contactos repetidos.

Na classe *Agenda* define ainda os métodos/construtores seguintes:

```

Agenda(string identificação);
int total(); //número total de contactos
void addContacto(Contacto c); //adicionar novo contacto
void addContacto(string nome, int telemóvel); //adicionar novo contacto
void remove(string nome); //remover contacto
bool existe(string nome); //determinar existência de contacto

```

```

bool existe(Contacto c); //determinar existência de contacto

int obterContacto(string nome); //obter telemóvel

string obterNomes(int telemóvel); //obter nome

bool igual(Agenda a) ; //igualdade entre duas agendas

Agenda ordena(); //ordenar agenda

```

3. Defina uma classe de teste contendo duas agendas.
4. Crie um menu interativo para manipular as agendas.

## Exercício 2:

Defina um sistema para gestão de um stand automóvel. Para cada automóvel o sistema deverá armazenar os dados seguintes:

- matrícula
- marca
- modelo
- cilindrada
- quilómetros
- preço
- estado (a venda ou vendido)

Cada cliente do stand deverá ficar armazenado no sistema com a informação seguinte:

- número de bilhete de identidade
- nome
- morada
- lista das matrículas dos veículos comprados no stand

Desenvolva uma aplicação com as funcionalidades seguintes:

- inserção de um novo automóvel no stand
- registo da venda de um automóvel

A informação dos automóveis deverá ser armazenada numa *SortedList*. Os dados dos clientes devem ser inseridos numa *ArrayList*, guardando as matrículas dos veículos comprados numa *Queue*.

## Referências

<http://www.softsteel.co.uk/tutorials/cSharp/>

<http://www.csharp-station.com/Tutorial.aspx>

<http://csharpcomputing.com/Tutorials/TOC.htm>

[http://msdn2.microsoft.com/en-us/library/9b9dty7d\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/9b9dty7d(VS.80).aspx)

[http://msdn2.microsoft.com/en-us/library/2s05feca\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/2s05feca(VS.80).aspx)

C# School – Programmers Heaven (cf. Site da disciplina)

## Parte VII – Herança

Autor: Luís Ferreira

---

Esta parte do documento C# Essencial, dá início ao estudo da Herança, Polimorfismo, Interfaces e Classes Abstractas.

### Alguma Terminologia

Alguns termos que convém conhecer:

- |                    |   |
|--------------------|---|
| <b>Atributo</b>    | uma variável que faz parte do estado de uma classe (o mesmo que campo)  |
| <b>Método</b>      | uma função que faz parte do comportamento de uma classe.  |
| <b>Constructor</b> | um método especial que é evocado quando é criada uma instância (um objecto) da classe.  |
| <b>Superclasse</b> | A classe da qual se herda (classe pai)  |
| <b>Subclasse</b>   | A classe herdada (classe filho)   |
| <b>Extende</b>     | uma classe estende outra se derivar (ou herdar) dela. Em Java usa-se a palavra <i>extends</i> .   |
| <b>Overload</b>    | um método diz-se overloaded se existem dois ou mais métodos com o mesmo nome na mesma classe. Cada método overloaded tem um conjunto de parâmetros diferentes.  |
| <b>Override</b>    | um método diz-se overridden se existe um método na sub-class com o mesmo nome e com os mesmos parâmetros (e naturalmente com implementação diferente). O método da superclass deixa assim de ser herdado. |

### Herança em POO

As Linguagens Orientadas a Objectos - POO (nomeadamente as modernas), referem três capacidades que suportam o desenho (*design*), estrutura (*structure*) e reutilização (*reusability*) de código. São elas:

- Encapsulamento
- Herança

- Polimorfismo

São também comuns os termos Módulo, Reutilização, Extensão e Derivação. Reutilizar deve ser visto como a possibilidade de utilizar um módulo (um componente, uma classe, um método, etc.) em aplicações diferentes, sem nenhuma alteração (ou com muito poucas) do seu código fonte. Extender, por outro lado, pode ser visto como a capacidade de “enriquecer” um módulo com algo (métodos, atributos, etc.), no sentido de o tornar mais adequado a um novo contexto. Em POO, a *Reutilização* deve ser vista no sentido de evitar a proliferação desnecessária de classes (porventura muitas acabam por fazer o mesmo que outras já existentes) e Extensão é suportada pelo mecanismo de *Herança* das classes para as suas sub-classes.

Como vimos no início deste documento, uma classe define um Tipo, uma “Regra” ou um “Template” a partir do qual é possível criar várias Instâncias (i.e., vários casos particulares). A Figura 12 mostra três instâncias da classe Veículo.

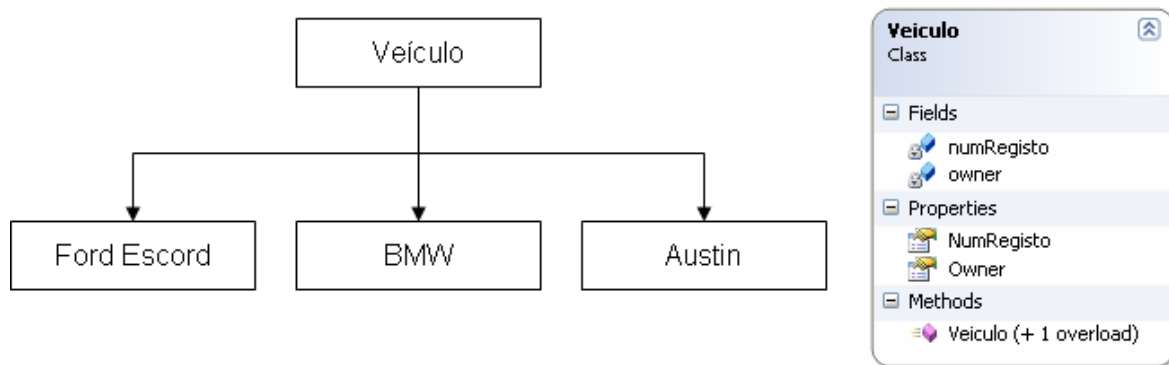


Figura 12 – Classes vs Instâncias

Se uma classe resulta da herança (*Inheritance* na bibliografia inglesa) de outra, então possui (“herda”) todos os seus membros de classe (estado e comportamento). À classe original dá-se, geralmente, o nome de *classe base* (classe pai ou super-classe) e à classe que herda chama-se classe filho, *sub-classe*, classe derivada ou classe herdada. No Diagrama de Classes<sup>6</sup> da Figura 13, as classes Carro, Mota e Tractor são sub-classes da classe Veículo.

<sup>6</sup> Diagrama gerado pelo Visual Studio C#

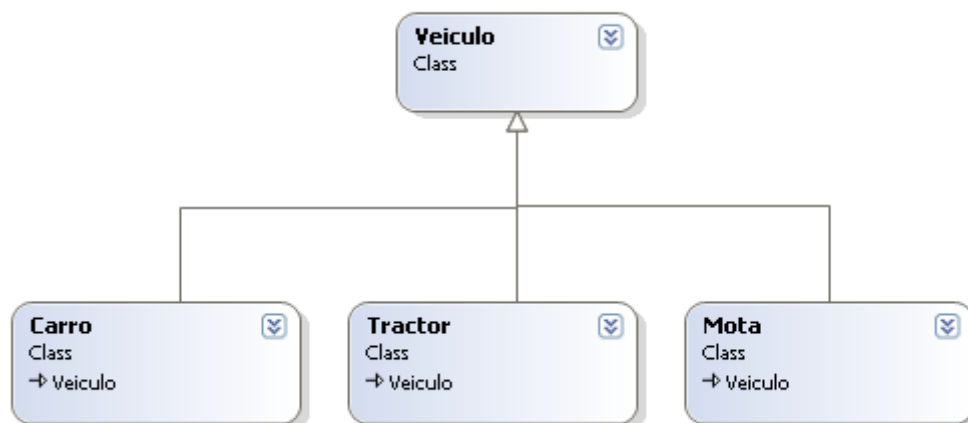


Figura 13 – Herança de classes

A Figura 14 mostra a herança das mesmas classes através de um Diagrama de Classes UML (Unified Modelling Language<sup>7</sup>). Tal como anteriormente, a classe base é Veiculo, enquanto que Carro, Tractor e Mota são sub-classes. A classe base geralmente possui comportamento global enquanto que as classes filho possuem comportamento específico. Daqui muitas vezes se dizer que em herança se vai *de uma Generalização para uma Especialização*.

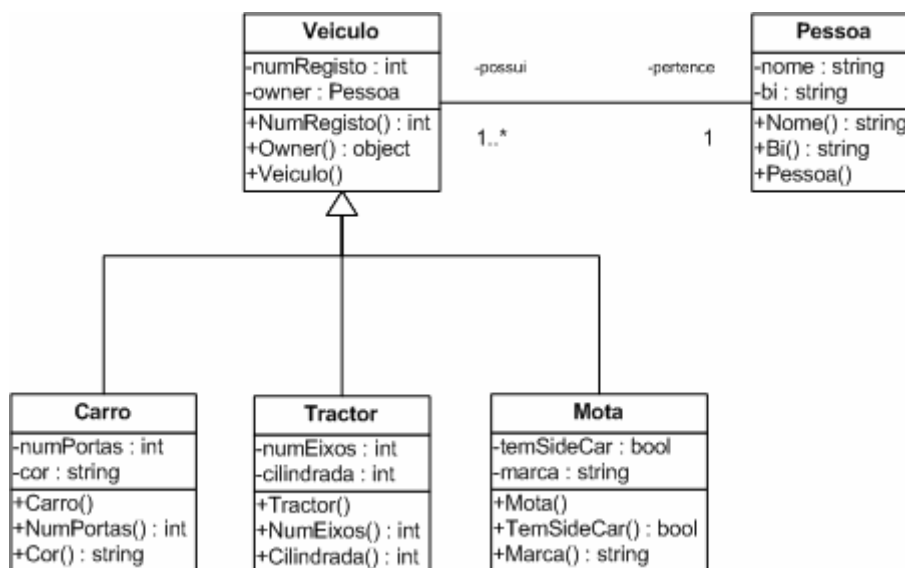


Figura 14 – Diagrama UML de Classes

<sup>7</sup> <http://www.omg.org/technology/documents/formal/uml.htm>



Assim, as classes Carro, Tractor e Mota herdam todos os atributos e métodos públicos da classe Veículos. No nosso caso, os atributos *numRegisto* e *owner* e os métodos *NumRegisto* e *Owner*.

É possível uma classe pai criar uma sua instância a partir da classe filho. Mas o contrário não. Ou seja, é possível

```
Veiculo v = new Carro("Preto", "lufer");
```

mas não é possível

```
Carro c = new Veiculo();
```

Por outro lado, é permitido

```
Console.WriteLine("Dono= "+v.Owner.Nome+" Registo="+ v.NumRegisto);
```

mas não é permitido

```
int aux = v.NumPortas;
```

Antes de passarmos a analisar a herança em C#, algumas comparações com o que se passa relativamente a outras linguagens orientadas a objectos, nomeadamente o JAVA:

Tal como no JAVA, em C# não é possível múltipla-herança. Em C++ é possível.

- A classe Object é a última classe base (ou super-classe) de todas as classes.
- Em C#, tal como em JAVA, é possível “muitipla-herança” de *interfaces*.
- Estruturas (*structures*) em C# só podem herdar (e implementar) interfaces. Não podem ser herdados.
- Deve dizer-se que uma *classe implementa um interface* versus uma classe deriva (ou herda) de um interface. A palavra herança deve utilizar-se só entre classes.

Vejamos, comparativamente, alguns exemplos de código:

## Herança em C++

```

class vehicle //class base em C++
{
protected:
    char colorname[20];
    int number_of_wheels;
public:
    vehicle();
    ~vehicle();
    void start();
    void stop();
    void run();
};

class Car: public vehicle //class derivada em C++
{
protected:
    char type_of_fuel;
public:
    Car();
};

```

## Herança em JAVA

```

class Vehicle {
    int number_of_wheels;
    private boolean accessories; //Anota se tem ou não acessórios
    protected String colorname;
    private static int counter;
    // Constructor
    Vehicle() {
        number_of_wheels = 5;
        accessories = true;
        colorname = "red";
        counter++;
    }
    // Instance methods
    public boolean isPresent() {
        return accessories;
    }
    private void getColor() {
        System.out.println("Cor do Veículo: " + colorname);
    }
    // Static methods
    public static void getNoOfVehicles() {
        System.out.println("Número de rodas: " + counter);
    }
}

```

```

class Car extends Vehicle {
    private int carNo = 10;
    public void printCarInfo() {
        System.out.println("Número: " + carNo);
        System.out.println("Número de rodas: " + number_of_wheels);
        // System.out.println("accessories: " + accessories); // Não herdado
        System.out.println("Accessórios: " + isPresent()); // Método Herdado.
        System.out.println("Cor: " + colorname); // Herdado.
        // System.out.println("Total de Veículos: " + counter); // Não herdado.
        getNoOfVehicles(); // Método Herdado
    }
}

public class VehicleDetails {

    public static void main(String[] args) {
        new Car().printCarInfo();
    }
}

```

## Herança em C#

Em C# utiliza-se o operador ":". A sintaxe normal de indicar a herança entre duas classes é:

```

class X : Y
{
    //corpo da classe
}

```

Que deve ser interpretado como, a classe X (classe filho) deriva da classe Y (classe pai).

Por exemplo, ao definirmos

```

class Gestor : Empregado
{
    //corpo da classe
}

```

Significa que a classe Gestor deriva da classe Empregado. Analisemos este exemplo com mais detalhe.

## Exemplo: Empregado → Gestor → Supervisor

Considere-se a classe Empregado definida por:

```

class Empregado
{
    //Atributos
    private string nome;
    private double custoHora;

    //Construtores
    public Empregado() { }

    public Empregado(string n, double v)
    {
        nome = n;
        custoHora = v;
    }

    //Properties
    public string Nome
    {
        get { return (nome); }
        set { nome = value; }
    }

    public double CustHora
    {
        get { return custoHora; }
        set { custoHora = value; }
    }

    //Métodos
    public double CalcSalario(int horas)
    {
        return (horas * custoHora);
    }
}

```

O estado desta classe é composto por duas variáveis (ou atributos) de instância privadas. Foram definidos dois construtores e duas propriedades, uma para cada variável do estado. Definiu-se também um método – CalcSalario – para calcular o salário a pagar.

Vamos agora analisar uma classe para descrever um Gestor. Um Gestor pode ser visto como um normal empregado mas com um vencimento diferente do de um funcionário. Por exemplo, vamos assumir que o funcionário é pago de acordo com as horas que trabalha e o Gestor tem um vencimento certo mensal.

```

class Gestor
{
    //Atributos
    string nome;
    double custoHora;
    bool assalariado;

    //Construtores
    public Gestor() { }

    public Gestor(string n, double v, bool ass)...

    //Properties
    public string Nome...

    public double CustHora...

    public bool Assalariado
    {
        get { return assalariado; }
        set { assalariado = value; }
    }

    //Métodos
    public double CalcSalario(int horas)
    {
        if (assalariado) return custoHora;
        return (horas * custoHora);
    }
}

```

De uma análise comparativa às duas classes (Empregado e Gestor) vimos que apenas um atributo (*assalariado*) e um método (*Assalariado*) as distingue. Igualmente se consegue verificar que alguns métodos comuns são ligeiramente implementados de forma diferente.

Até a forma como podem ser utilizadas são idênticas:

```

Empregado e = new Empregado("Manuel", 1200);
Gestor g = new Gestor("Joaquim", 1200, true);

```

Resumindo, foi necessário escrever uma nova classe cuja definição é praticamente idêntica à de um Empregado. Terá sido trabalho desnecessário?

## Reutilização

Vamos tentar agora definir um Gestor a partir da definição de um Empregado – *Reutilização*. Ao derivar de Empregado, o Gestor herda todos os seus dados e comportamento. É possível depois acrescentar novos dados ou métodos específicos para o Gestor e redefinir eventuais métodos existentes. Assim, a nova classe de Gestor poderia ficar:

```

class Gestor : Empregado
{
    private bool assalariado;

    //Construtor
    public Gestor(string n, double v, bool b)
        : base(n, v)
    {
        assalariado = b;
    }

    //Properties
    public bool Assalariado
    {
        get { return assalariado; }
        set { assalariado = value; }
    }

    //Métodos
    public new double CalcSalario(int horas)
    {
        if (assalariado) return custoHora;
        return (horas * custoHora);
    }
}

```

Repare-se que muito do código não necessitou de ser reescrito, ie, foi herdado.

Uma vez que os construtores não são explicitamente herdados, é necessário redefinir qualquer construtor com parâmetros. No nosso caso foi necessário redefinir o construtor Gestor. O seu “trabalho” é parte do mesmo que é feito pelo construtor da classe Empregado. Por isso, reutilizou-se o construtor dessa classe pai. Tal é feito através do operador *base* (tal como em C++), tal como se observa na sua definição:

```

//Construtor
public Gestor(string n, double v, bool b) : base(n, v)
{
    assalariado = b;
}

```

Repare-se que o parâmetro *b* é usado localmente mas os parâmetros *n* e *v* são passados para o construtor Empregado através da instrução *base(n,v)*.

Isto entende-se também atendendo ao facto de que, sempre que se cria uma instância de uma sub-classe, em primeiro lugar é criado uma instância da classe pai.

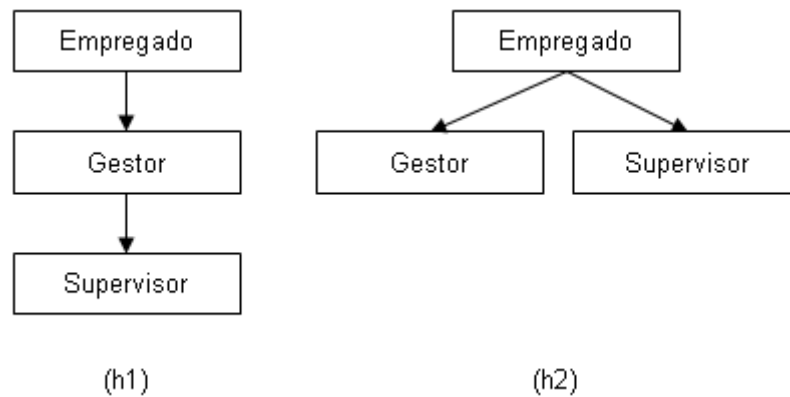
Sempre que um método da classe filho pretenda utilizar um método ou atributo público da classe pai, tal é feito através do operador *base* (equivalente ao operador *super* em JAVA). Por exemplo, o método *ShowData* da classe Gestor poderia ser implementado da seguinte forma:

```

public void ShowData()
{
    Console.WriteLine("Nome: " + base.Nome);
    Console.WriteLine("Assalariado: " + assalariado);
    Console.WriteLine("Valor: " + CalcSalario(3));
}

```

Se a ideia fosse criar uma nova classe para um *Supervisor*, poderíamos avançar com duas hipóteses: ou se trata de um Gestor (h1) ou de um Empregado (h2).



Estaríamos assim a falar de:

```

class Supervisor : Empregdo
{
}

```

ou

```

class Supervisor : Gestor
{
}

```

## Mais conceitos

### Membros *Protected*

Sempre que um membro (atributo ou método) é classificado como *protected*, só pode ser utilizado dentro da classe pai ou classe filho. Não pode ser usado através de qualquer instância dessas classes.

```

class X
{
    protected void ShowValues()
    {
    }
}

class Y : X
{
    public void ShowValuesY(){
        base.ShowValues();    //aqui é permitido
    }
}

static void Main()
{
    X aux = new X();
    aux.ShowValues();        //aqui não é permitido
}

```

Um membro *protected* deve ser visto como *private* para as instâncias dessa classe.

### Classes *sealed*

Caso se pretenda evitar que uma classe seja herdada, utiliza-se o classificador *sealed*. Assim, no seguinte a herança não será permitida, uma vez que a class X está definida como *sealed*.

```

sealed class X
{
}

class Y : X
{
}

```

### Exemplos completos

#### Exemplo 1: Estudante → EstudanteVIP

Dois tipos de estudantes Normal e VIP. Para cada estudante é necessário conhecer:

- Número



- Nome
- Data de Nascimento
- É necessário também
- Calcular a idade do aluno
- Mostrar os dados pessoais do aluno

Para o estudante VIP é necessário também guardar a informação sobre a média final obtida.

### **Classe Student**

```
//by lufer - IPCA
//POO - 2008-2008
using System;
using System.Text;
/// <summary>
/// Classe Student
/// </summary>
/// <remarks>
/// Demonstrar a aplicação de Herança de Classes
/// </remarks>
class Student
{
    //atributos Privados
    private int num;
    private string nome;
    private DateTime data_nasc;
    //Construtores
    public Student()
    {}
    public Student(int cod, string n, DateTime nasc)
    {
        num = cod;
        data_nasc = nasc;
        nome = n;
    }

    //Properties
    // Só permite ler
    public int Num
    {
        get { return num; }
    }

    public DateTime Data_Nasc
    {
        get { return data_nasc; }
        set { data_nasc = value; }
    }

    public string Nome
    {
```

```

    get { return nome; }
    set { nome = value; }
}

//Métodos
/// <summary>
/// Devolve a idade do aluno
/// </summary>
/// <returns>int</returns>
public int GetAge()
{
    int age = DateTime.Now.Year - data_nasc.Year;
    return age;
}

/// <summary>
/// protected method: só pode ser usado dentro desta classe ou
/// dentro das suas sub-classes
/// </summary>
protected void ShowStudent()
{
    Console.WriteLine("Nome= {0} - Idade={1}", nome, num);
}
}

```

### Classe StudentVIP

```

/// <summary>
/// Herda toda a informação de Student e acrescenta mais
/// </summary>
class StudentVIP : Student
{
    //atributos Private
    private double media;

    //Construtores

    /// <summary>
    /// Versão 1
    /// </summary>
    public StudentVIP()
    {}

    /*
    /// <summary>
    /// Versão 2 com base. Reutiliza o construtor da classe pai
    /// </summary>

    public StudentVIP(): base()
    {}
    */

    /// <summary>
    /// Construtor: manipula a média do aluno
    /// </summary>

```

```

    /// <param name="m">media</param>
    public StudentVIP(int m)
    {
        media = m;
    }

    /// <summary>
    /// Versão 1: uso explicito das Propriedades da classe Pai
    /// </summary>
    /// <param name="cod">numero do aluno</param>
    /// <param name="m">media</param>
    /// <param name="n">nome</param>
    /// <param name="dn">data de Nascimento</param>
    public StudentVIP(int cod, string n, DateTime dn, int med)
    {
        this.media = med;
        this.Data_Nasc = dn;
        this.Nome = n;
        //this.Num = cod; //Incorrecto. Propr. Num só permite Get
    }

    /// <summary>
    /// Versão 2: uso explicito do construtor da classe Pai
    /// </summary>
    /// <param name="cod">numero do aluno</param>
    /// <param name="med">Media</param>
    /// <param name="n">Nome</param>
    /// <param name="dn">Data de Nascimento</param>
    /// <remarks>
    /// Os valores cod, n e dn são passados para o construtor pai
    /// </remarks>
    /// public StudentVIP(int cod, string n, DateTime dn, int med) :
    /// base(cod,n,dn)
    /// {
    /// this.media = med;
    /// }

    //Outros Métodos
    public string Classificacao()
    {
        string med;
        if (media < 10) med = "Reprovado";
        else
            med = "Aprovado";
        return med;
    }

    //Usa protected method dentro da sub-classe
    public void ShowStudentVIP()
    {
        base.ShowStudent(); //""base"" equivale ao super() do JAVA
    }
}

```

**Pequena demonstração para “Linha de comando”:**

```

private static void Main()
{
    //instância de Student
    Student s1 = new Student(12, "Manuel", new DateTime(1980,12,12));
    Console.WriteLine("S1= " + s1.Num + " Idade=" + s1.GetAge().ToString());

    //instância de StudentVIP
    StudentVIP sv1 = new StudentVIP(17);
    sv1.Nome = "JJ";
    sv1.Data_Nasc = new DateTime(1982, 03, 04);
    Console.WriteLine ("SV1= " + sv1.Num + " Idade=" +
        sv1.GetAge().ToString() + " Estado:" + sv1.Classificacao());

    //instância de StudentVIP
    StudentVIP sv2 = new StudentVIP(3245, "Joaquim", new DateTime(1987,
        01, 01),9);
    Console.WriteLine ("SV2= " + sv2.Num + " Idade=" +
        sv2.GetAge().ToString()+" Estado:"+sv2.Classificacao());

    sv2.ShowStudentVIP();

    //sv2.ShowStudent(); //Não é possível...é protected
    //s1.ShowStudent(); //Não é possível...é protected

    Student s2 = new StudentVIP();           //Será possível?
    //s2.Classificacao();                     //Será possível? Não!
    s2.GetAge();                             //Será possível?
    sv2.GetAge();

    //StudentVIP sv3 = new Student();        //Será possível?
}

```

O resultado seria:

```

S1= 12 Idade=28
SV1= 0 Idade=26 Estado:Aprovado
SV2= 0 Idade=21 Estado:Reprovado
Nome= Joaquim - Idade=0
Prima qualquer tecla para continuar . . . _

```

### Exemplo2 : Empregado → Gestor → Supervisor

Neste exemplo surgem dois termos novos: virtual e override que serão analisados quando os Polimorfismos.

#### Classe Empregado

```

//by lufer - 2007-2008
//IPCA-EST
//POO
//lufer@ipca.pt
using System;
using System.Collections.Generic;
using System.Text;

namespace Resumo_Heranca

```

```

{
    /// <summary>
    /// Exemplo demonstrativo de Herança de classes em C#
    /// </summary>
    /// <remarks>
    /// Classes: Empregado -> Gestor --> Supervisor
    /// </remarks>
    class Empregado
    {
        //Atributos
        public string nome;
        public double custoHora;

        //Construtores
        public Empregado() { }

        public Empregado(string n, double v)
        {
            nome = n;
            custoHora = v;
        }

        //Properties
        public string Nome
        {
            get { return(nome); }
            set { nome = value; }
        }

        public double CustHora
        {
            get { return custoHora; }
            set { custoHora = value; }
        }

        //Métodos
        public virtual double CalcSalario(int horas)
        {
            return (horas * custoHora);
        }
    }
}

```

#### **Classe Gestor**

```

class Gestor : Empregado
{
    private bool assalariado;

    //Construtor

    public Gestor() { }

    public Gestor(string n, double v, bool b): base(n,v)
    {
        assalariado = b;
    }
}

```

```

//Properties
public bool Assalariado
{
    get { return assalariado; }
    set { assalariado = value; }
}

//Métodos
public override double CalcSalario(int horas)
{
    if (assalariado) return custoHora;
    return (horas * custoHora);
}

public virtual void ShowData()
{
    Console.WriteLine("Nome: " + base.Nome);
    Console.WriteLine("Assalariado: " + assalariado);
    Console.WriteLine("Valor:" + CalcSalario(3));
}
}

```

### **Classe Supervisor**

```

class Supervisor : Gestor
{
    double premio;

    public Supervisor()
    {
        premio = 0;
    }

    public Supervisor(int p, string n, double v, bool a)
        : base(n, v, a)
    {
        premio = p;
    }

    public override void ShowData()
    {
        base.ShowData();
        Console.WriteLine("Premio=" + premio);
    }
}

```

### **Pequena demonstração para “Linha de comando”:**

```

static void Main(string[] args)
{
    Empregado x = new Empregado("Manuel", 12);
    Gestor y = new Gestor("Jorge", 1200, true);

    y.ShowData();

    Supervisor s = new Supervisor(200, "Ana", 1200, false);
}

```

```
s.ShowData();
}
```

Resultado:

```
Nome: Jorge
Assalariado: Sim
Valor:1200
Nome: Ana
Assalariado: Não
Valor:3600
Premio=200
```

## Polimorfismo

Polimorfismo<sup>8</sup> (*várias formas*) representa a capacidade de uma classe reescrever a implementação de um método existente, i.e., implementar de forma diferente o mesmo método. Desta forma contribui-se também para o encapsulamento, evitando ao utilizador o conhecimento da forma como está implementado.

Consideremos o seguinte exemplo de uma hierarquia (especialização de *DrawingObject*) com o mesmo método (*Draw*) a surgir em cada nova classe.

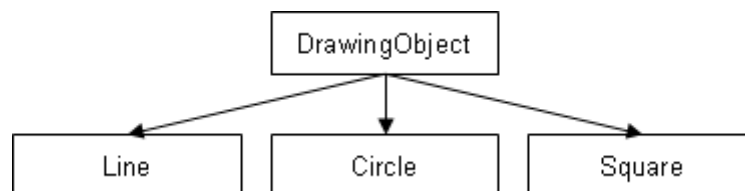


Figura 15 - Especialização de uma classe

```
/// <summary>
/// Polimorfismo
/// </summary>
public class DrawingObject
{
    public void Draw()
    {
        Console.WriteLine("Desenho genérico....");
    }
}

public class Line : DrawingObject
{
    public void Draw()
    {
        Console.WriteLine("Linha.");
    }
}
```

<sup>8</sup> Não confundir com *Overloading*

```

    }
}

public class Circle : DrawingObject
{
    public void Draw()
    {
        Console.WriteLine("Círculo.");
    }
}

public class Square : DrawingObject
{
    public void Draw()
    {
        Console.WriteLine("Quadrado.");
    }
}

static void Main(string[] args)
{
    DrawingObject[] dObj = new DrawingObject[4];

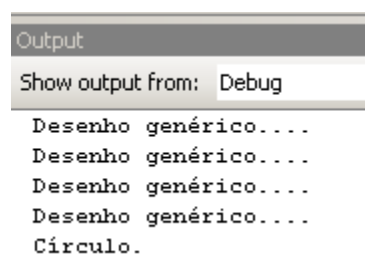
    dObj[0] = new Line();
    dObj[1] = new Circle();
    dObj[2] = new Square();
    dObj[3] = new DrawingObject();

    foreach (DrawingObject drawObj in dObj)
    {
        drawObj.Draw();
    }

    Circle c = new Circle();
    c.Draw();
}

```

Embora o compilador produza vários *warnings*<sup>9</sup>, o resultado será



```

Output
Show output from: Debug
Desenho genérico....
Desenho genérico....
Desenho genérico....
Desenho genérico....
Círculo.

```

<sup>9</sup> Warning 1'Line.Draw()' hides inherited member 'DrawingObject.Draw()'. Use the new keyword if hiding was intended.



Repare-se que, embora se estejam a criar aparentemente instâncias de Line, Circle e Square, na realidade estão a criar-se instâncias de DrawingObject. A mensagem “*Desenho genérico*” é prova disso. O método Draw que está a ser executado é sempre o da classe base.

### Reescrever Métodos – *override*

Caso se pretenda reescrever os métodos em cada classe filho, por exemplo, para uma implementação diferente, é necessário utilizar os classificadores *virtual* nos métodos da classe pai e *override* na reescrita do método na classe filho.

Se analisarmos de novo o nosso exemplo, agora com reescrita dos métodos,

```
/// <summary>
/// Polimorfismo
/// </summary>
public class DrawingObject
{
    public virtual void Draw()
    {
        Console.WriteLine("Desenho genérico....");
    }
}

public class Line : DrawingObject
{
    public override void Draw()
    {
        Console.WriteLine("Linha.");
    }
}

public class Circle : DrawingObject
{
    public override void Draw()
    {
        Console.WriteLine("Círculo.");
    }
}

public class Square : DrawingObject
{
    public override void Draw()
    {
        Console.WriteLine("Quadrado.");
    }
}

static void Main(string[] args)
{
    DrawingObject[] dObj = new DrawingObject[4];

    dObj[0] = new Line();
```

```

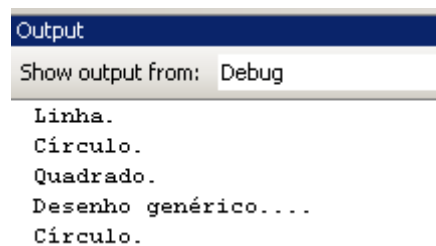
dObj[1] = new Circle();
dObj[2] = new Square();
dObj[3] = new DrawingObject();

foreach (DrawingObject drawObj in dObj)
{
    drawObj.Draw();
}

Circle c = new Circle();
c.Draw();
}

```

o resultado seria



```

Output
Show output from: Debug
Linha.
Círculo.
Quadrado.
Desenho genérico....
Círculo.

```

e como se pode verificar pelas mensagens resultantes, são evocados os métodos Draw de cada sub-classe. Isto acontece porque o método Draw da classe pai é virtual e como tal, o compilador procura em primeiro lugar a definição do método na classe referenciada<sup>10</sup>.

## Operador *new*

Caso se pretenda uma nova implementação de um método da classe base, sem contudo o reescrever (*override*), é necessário utilizar o operador *new*. Evita-se assim o polimorfismo.

Suponhamos que se pretendia redefinir um método Draw() na classe Circle

```

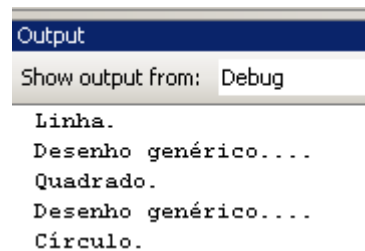
public class Circle : DrawingObject
{
    public new void Draw()
    {
        Console.WriteLine("Círculo.");
    }
}

```

Neste caso, o resultado seria

---

<sup>10</sup> Contrariamente ao JAVA e tal como em C++, os métodos são não-virtuais por omissão. Contrariamente ao JAVA e ao C++, é necessário definir de forma explícita o método *override*.



```
Output
Show output from: Debug
Linha.
Desenho genérico....
Quadrado.
Desenho genérico....
Círculo.
```

Uma vez que, ao não haver polimorfismo na classe Circle, o compilador procura o método Draw da classe pai.

Para conseguir reescrever o método da sub-classe, utiliza-se o operador *override*. Para criar uma nova implementação do método utiliza-se o operador *new*.

## Classes Abstratas

Classes abstratas são classes incompletas, i.e., existe algum método da classe que não se encontra implementado. Geralmente numa classe abstrata verifica-se só a assinatura dos métodos que se pretendem abstratos. A implementação desses métodos será feita numa sua sub-classe. Por ser incompleta, não é possível criar uma instância de uma classe abstrata.

Quando uma classe deriva de uma classe abstrata e implementa todos os seus métodos, trata-se de uma classe concreta. Uma classe abstrata é definida através do classificador *abstract*.

Uma classe abstrata pode e deve ter construtores definidos!

Analisemos o seguinte exemplo:

```

/// <summary>
/// Classe abstrata para uma Calculadora
/// </summary>
abstract class Calculadora
{
    /// <summary>
    /// Construtor de classe abstrata: deve ser protected
    /// </summary>
    protected Calculadora()
    {
    }

    //propriedade abstrata
    public abstract int X
    {
        get;
        set;
    }

    //metodo abstrato
    public abstract int Soma();

    /// <summary>
    /// Implementa a Subtração de dois números
    /// </summary>
    /// <param name="x"></param>
    /// <param name="y"></param>
    /// <returns></returns>
    public int Sub(int x, int y)
    {
        return (x - y);
    }
}

```

Trata-se de uma classe abstrata pois existe um método que não está implementado (*Add*). Assim sendo, não é permitido criar uma instância (um objeto) desta classe:

```
Calculadora x = new Calculadora();    //inválido
```

Para o conseguir fazer, é necessário definir uma sub-classe de *Calculadora*. Por exemplo,

```

/// <summary>
/// Classe concreta: implementa a classe abstracta
/// </summary>
class BoaCalculadora : Calculadora
{
    int x, y;

    /// <summary>
    /// Propriedade herdada
    /// </summary>
    public override int X
    {
        get { return x; }
        set { x = value; }
    }

    /// <summary>
    /// Nova propriedade
    /// </summary>
    public int Y
    {
        get { return y; }
        set { if (value > 0) y = value; }
    }

    public override int Soma()
    {
        return x + y;
    }
}

```

Agora já é possível o seguinte código:

```
BoaCalculadora aux = new BoaCalculadora();
```

A ideia de uma classe abstrata assenta na possibilidade de definir um conjunto de “regras a seguir”, tipo uma classe modelo, e permitir que alguns dos métodos possam ter implementações diferentes em processos de herança.

## Interfaces

Podem ser vistos como um tipo de dados particular, semelhante às classes abstratas, mas onde nenhum método é implementado mas simplesmente especificado, i.e., só existe a assinatura dos métodos. Funciona de facto como uma classe modelo à qual todas as classes que o seguem (que o implementam) terão de respeitar na íntegra.

Um interface é implementado por uma classe. A classe que o implementa tem de definir a implementação de todos os métodos especificados no interface.

Uma classe pode implementar mais do que um Interface, contrariamente à herança que tem de ser simples. Define-se um interface através do classificador *interface* e convencionou-se que em C# dos interfaces devem começar por “I”.

Vejamos um exemplo possível para um interface para a nossa calculadora:

```
interface ICalculadora
{
    int TotOpera
    {
        get;
        set;
    }
    int Add(int x, int y);
    int Sub(int x, int y);
}
```

Agora é possível definir a uma nova classe que implementa o interface *ICalculadora*:

```
class Calculadora2 : ICalculadora
{
    int tot;

    public int TotOpera
    {
        get { return tot; }
        set { tot = value; }
    }

    public int Add(int x, int y)
    {
        return (x + y);
    }

    public int Sub(int x, int y)
    {
        return (x - y);
    }
}
```

Repare-se como todos os métodos são definidos e obrigatoriamente classificados como públicos.

## Múltiplos Interfaces

Como já vimos antes, uma classe pode implementar mais do que um interface. Neste caso deve definir a implementação de todos os métodos de todos os interfaces que implementa.

Consideremos o seguinte interface

```
interface IMessage
{
    void ShowMsg(string s);
    string ReadValue();
}
```

Suponhamos agora que a nossa classe implementa os interfaces *IMessage* e *ICalculadora*. Assim poderia ser definida por:

```
class Calculadora3 : ICalculadora, IMessage
{
    int tot;

    public int TotOpera
    {
        get { return tot; }
        set { tot = value; }
    }

    public int Add(int x, int y)
    {
        return (x + y);
    }

    public int Sub(int x, int y)
    {
        return (x - y);
    }

    public void ShowMsg(string s)
    {
        Console.WriteLine(s);
    }

    public string ReadValue()
    {
        return (Console.ReadLine());
    }
}
```

Repare-se no início da definição da classe a identificação dos interfaces que se pretendem implementar:

```
class Calculadora3 : ICalculadora, IMessage
```

Analisemos outro exemplo:

```

/// <summary>
/// Interface para Pessoa
/// </summary>
interface IPessoa
{
    string Nome
    {
        get;
        set;
    }

    string UserID
    {
        get;
        set;
    }

    string GetPass();
    string GetUserId();
}

```

A classe Pessoa que implementa *IPessoa*:

```

/// <summary>
/// Classe que descreve uma Pessoa e respeita o interface IPessoa
/// </summary>
class Pessoa : IPessoa
{
    string nome;
    string userID;
    string passwd;

    public string Nome
    {
        get{return nome;}
        set{nome = value;}
    }

    public string UserID
    {
        get { return userID; }
        set { userID = value; }
    }

    public string GetUserId()
    {
        return "";
        //throw new NotImplementedException();
    }

    public string GetPass()
    {
        return "";
    }
}

```



E a classe *Coisa* também implementa *IPessoa*:

```

/// <summary>
/// Classe Coisa que também implementa IPessoa
/// </summary>
class Coisa : IPessoa
{
    string nome;
    string userID;
    string passwd;

    public string Nome
    {
        get { return nome; }
        set { nome = value; }
    }

    public string UserID
    {
        get { return userID; }
        set { userID = value; }
    }

    public string GetUserId()
    {
        return "";
    }

    public string GetPass()
    {
        return "";
    }
}

/// <summary>
/// Outro Interface
/// </summary>
interface IRandomNumberGen
{
    int GetNextNumber();
}

```

A classe *OutraIII* implementa os dois interfaces: *IRandomNumberGen* e *ICalculadora*

```

class OutraIII : IRandomNumberGen, ICalculadora
{
    /// <summary>
    /// Gerar números aleatórios entre 0 e um máximo
    /// </summary>
    /// <returns></returns>
    public int GetNextNumber()...

    /// <summary> ...
    public int GetNextNumber(int min, int max)
    {
        return (new Random()).Next(min, max);
    }

    /// <summary> ...
    public int Soma(int x, int y)
    {
        return x + y;
    }

    /// <summary> ...
    public int Abs(int x)
    {
        return ((x < 0) ? (-1 * x) : x);
    }
}

```

## Herança de Interfaces

Tal como em classes, é possível definir herança em interfaces assim como condicionar a conversão (*cast*) entre interfaces.

Analise-se o seguinte exemplo. A classe *User* implementa o interface *IUser* e possui um método que recebe como parâmetro um objeto que respeite o interface *IPessoa*.

```

/// <summary>|
/// Interface
/// </summary>
interface IUser
{
    bool Login(IPessoa p);
}

```

```

/// <summary>
/// Class que descreve utilizador
/// </summary>
class User: IUser
{
    public bool Login(IPessoa p)
    {
        return String.IsNullOrEmpty(p.Nome);
    }

    //O mesmo que

    //public bool Login(Pessoa p)
    //{
    //    return String.IsNullOrEmpty(p.Nome);
    //}

    //public bool Login(Coisa c)
    //{
    //    return String.IsNullOrEmpty(c.Nome);
    //}
}

```

Assim, é possível utilizar o método Login com qualquer objeto que respeite o interface IPessoa.

```

Pessoa p = new Pessoa();
Coisa c = new Coisa();
User u = new User();

```

É possível invocar o método Login com um objeto tipo Pessoa (p).

```
bool aux = u.Login(p);
```

Ou é possível invocar o método Login com um objeto tipo Coisa (c).

```
aux = u.Login(c);
```

## Interfaces vs Classes abstratas

**AQUI**

Feature	Interface	Abstract class
Multiple inheritance	A class may inherit several interfaces.	A class may inherit only one abstract class.

Feature	Interface	Abstract class
Default implementation	An interface cannot provide any code, just the signature.	An abstract class can provide complete, default code and/or just the details that have to be overridden.
Access Modifiers	An interface cannot have access modifiers for the subs, functions, properties etc everything is assumed as public	An abstract class can contain access modifiers for the subs, functions, properties
Core VS Peripheral	Interfaces are used to define the peripheral abilities of a class. In other words both Human and Vehicle can inherit from a IMovable interface.	An abstract class defines the core identity of a class and there it is used for objects of the same type.
Homogeneity	If various implementations only share method signatures then it is better to use Interfaces.	If various implementations are of the same kind and use common behaviour or status then abstract class is better to use.
Speed	Requires more time to find the actual method in the corresponding classes.	Fast
Adding functionality (Versioning)	If we add a new method to an Interface then we have to track down all the implementations of the interface and define implementation for the new method.	If we add a new method to an abstract class then we have the option of providing default implementation and therefore all the existing code might work properly.
Fields and Constants	No fields can be defined in interfaces	An abstract class can have fields and constants defined

## Exercícios

Desenhe uma hierarquia de classes capaz de descrever um livro (...→secções→capítulos→...). Crie uma pequena aplicação que a demonstre.

Desenhe uma hierarquia de classes capaz de descrever uma árvore genealógica (relação de ...Avós→Pais→Filhos....). Crie uma pequena aplicação que a demonstre.

Pretende-se desenhar uma aplicação para a Junta Autónoma de Estradas, capaz de gerir a informação sobre Veículos. O Diagrama de Classes a seguir representa a hierarquia de entidades a tratar. Procure implementá-la (e completá-la) em C#, sabendo que:

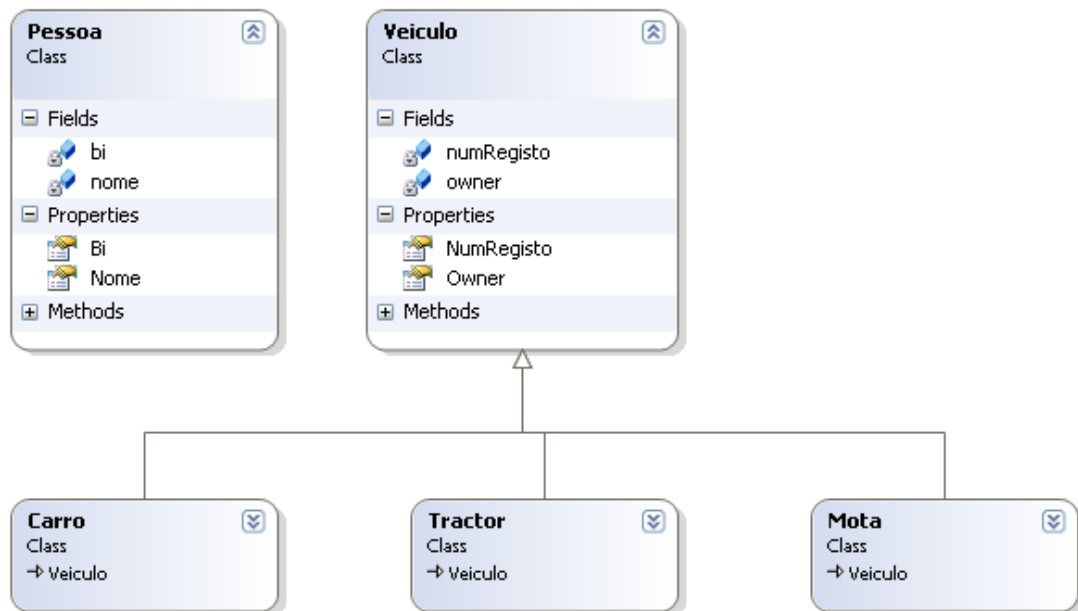


Figura 16 - Exercício sobre Herança de classes

- Todo o veículo tem uma pessoa como dono (owner);
- Um carro é um veículo com uma cor e um determinado numero de portas
- Uma moto é um veículo de uma determinada marca e deve ter ou não sidecar.
- Um tractor é um veículo com um determinado número de eixos e com um motor de uma determinada cilindrada.
- O número de registo de qualquer veículo é sequencial e único.

a) Faça uma pequena aplicação capaz de permitir executar o seguinte código Main()

```

public static void Main()
{
    Veiculo c = new Carro("Preto", "lufer");

    Console.WriteLine("Dono= " + c.Owner.Nome + " Registro=" + c.NumRegistro);

    Carro x = new Carro("Preto");

    Console.WriteLine(c.NumRegistro);

    //Console.WriteLine("Cor = " + c.Cor); É possível?
    Console.WriteLine("Cor= " + x.Cor);

    Veiculo m = new Mota();

    Console.WriteLine(m.NumRegistro);

    Console.WriteLine("Proprietario da Mota 1=" + m.Owner.Nome);

    Veiculo m1 = new Mota("Luis");
    Console.WriteLine("Proprietario da Mota 2=" + m1.Owner.Nome);
}

```

Desenhe um Interface que se aplique à classe Veiculo.

## Referências

- <http://www.softsteel.co.uk/tutorials/cSharp/>
- <http://www.csharp-station.com/Tutorial.aspx>
- <http://csharpcomputing.com/Tutorials/TOC.htm>
- [http://msdn2.microsoft.com/en-us/library/9b9dty7d\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/9b9dty7d(VS.80).aspx)
- [http://msdn2.microsoft.com/en-us/library/2s05feca\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/2s05feca(VS.80).aspx)
- C# School – Programmers Heaven (cf. Site da Disciplina)

## Parte VIII – Excepções

Autor: Marco Costa

---

Esta parte do documento C# Essencial dá início ao estudo do mecanismo de exceções da linguagem C#.

### Sumário:

## Introdução

O mecanismo de exceções da linguagem C# permite lidar com erros que ocorrem durante a execução de programas.

Vamos começar por estudar situações em que as exceções são úteis. De seguida, veremos como lidar com as exceções que são “lançadas” quando ocorrem determinados tipos de erros. Depois exploraremos as funcionalidades do mecanismo de exceções da linguagem C#. Por fim, aprenderemos a criar os nossos próprios tipos de exceções.

## Erros

Considere o seguinte código, destinado a escrever o resultado da soma de dois números passados como parâmetros para a aplicação (na linha de comandos).

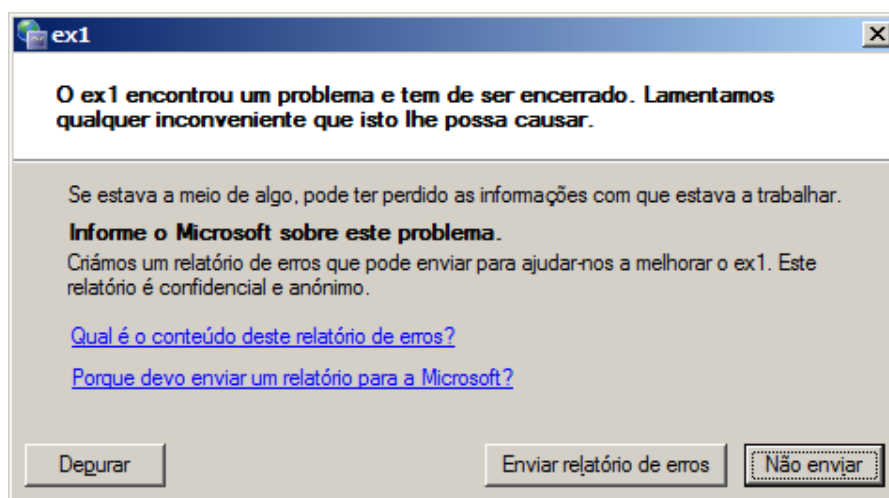
```
static void Main(string[] args)
{
    if (args.Length != 2)
        Console.WriteLine("Utilização: ex1.exe <num1> <num2>");
    else
    {
        int a = int.Parse(args[0]);
        int b = int.Parse(args[1]);
        int soma = a + b;
        Console.WriteLine("{0} + {1} = {2}", a, b, soma);
    }
}
```

Suponhamos que o utilizador invoca o programa da seguinte forma:

```
ex1.exe 255 abc
```

Uma vez que o segundo parâmetro não representa um número, ocorre um erro com que a aplicação não lida. Assim sendo, o sistema operativo (Windows XP) apresenta o seguinte diálogo, que informa o utilizador (de forma diplomática) que a aplicação “crashou”:





Na consola é escrita a seguinte mensagem de erro:

Unhandled Exception: System.FormatException: Input string was not in a correct format.

```
at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number,
NumberFormatInfo info, Boolean parseDecimal)
```

```
at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
```

```
at ex1.Program.Main(String[] args) in C:\Documents and Settings\Administrador\Os meus documentos\Visual
Studio 2005\Projects\chapter-exceptions\ex1\Program.cs:line 15
```

A mensagem de erro indica que foi lançada uma excepção quando o segundo parâmetro estava a ser convertido para um *int* com o método *Parse* (nota: falam algumas linhas no código apresentado).

Erros como este e muitos outros podem ocorrer durante a execução de aplicações. Nalguns casos, é possível incluir verificações no código para prevenir a ocorrência de erros, mas muitas vezes isso iria complicar excessivamente o código, e nunca seria possível prevenir todas as situações de erro. O mecanismo de excepções permite-nos lidar de forma elegante com este tipo de situações, sem ter que complicar demasiado o nosso código.

## Excepções

Em C#, quando ocorre um erro inesperado durante a execução de um programa (ou parte de um programa), a forma correcta de lidar com esse erro é lançar uma excepção. Essa excepção vai interromper a execução normal do programa, e se não for “apanhada”, vai levar à terminação da aplicação. Vejamos um exemplo que consiste na aplicação anterior com a adição de código para lidar com excepções:

```

static void Main(string[] args)
{
    if (args.Length != 2)
        Console.WriteLine("Utilização: ex1.exe <num1> <num2>");
    else
    {
        try
        {
            int a = int.Parse(args[0]);
            int b = int.Parse(args[1]);
            int soma = a + b;
            Console.WriteLine("{0} + {1} = {2}", a, b, soma);
        }
        catch
        {
            Console.WriteLine("Ocorreu um erro. Utilização: ex1.exe <num1> <num2>");
        }
    }
}

```

Tal como anteriormente, a aplicação recebe dois parâmetros, converte-os para números, e escreve na consola a sua soma. Suponhamos que o utilizador continua a invocar o programa da seguinte forma:

```
ex1.exe 255 abc
```

Uma vez que o segundo parâmetro continua a não ser um número, vai ocorrer um erro. Mas desta vez, vai aparecer a seguinte mensagem na consola:

```
Ocorreu um erro. Utilização: ex1.exe <num1> <num2>
```

A aplicação não crashou e conseguiu lidar com o erro de forma graciosa. Isso aconteceu porque o código tira proveito do mecanismo de exceções:

- O código em que podem ocorrer erros é contido num bloco *try*, seguido de um bloco *catch*. Isso faz com que se for lançada uma exceção dentro do bloco *try*, o fluxo normal de execução é interrompido, passando a ser executado o código contido no bloco *catch*.
- O código dentro do bloco *catch* escreve uma mensagem de erro na consola, e a execução continua no código a seguir aos blocos *try* e *catch*. Neste caso, como não existe mais código, a aplicação termina (sem crashar).

No exemplo apresentado, a exceção foi lançada no código contido no bloco *try*,<sup>11</sup> mas a exceção seria apanhada no bloco *catch* mesmo se tivesse sido lançada noutra parte do código, desde que esse código tivesse sido invocado, directa ou indirectamente, dentro do bloco *try*. Vejamos um exemplo:

```
static void Somar(string arg1, string arg2)
{
    int a = int.Parse(arg1);
    int b = int.Parse(arg2);
    int soma = a + b;
    Console.WriteLine("{0} + {1} = {2}", a, b, soma);
}

static void Main(string[] args)
{
    if (args.Length != 2)
        Console.WriteLine("Utilização: ex1.exe <num1> <num2>");
    else
    {
        try
        {
            Somar(args[0], args[1]);
        }
        catch
        {
            Console.WriteLine("Ocorreu um erro.");
        }
    }
}
```

Neste exemplo, o erro ocorre no método *Somar*, e é aí que surge a exceção. No entanto, a exceção continua a ser apanhada dentro do bloco *catch* do método *Main*.

## Tipos de exceções

Durante a execução de uma aplicação podem ocorrer diversos tipos de erros, por isso é natural que existam diversos tipos de exceções. Em C# (e na plataforma .NET) cada exceção corresponde a uma classe, e as diversas exceções são organizadas numa hierarquia. A classe *Exception* corresponde à raiz da hierarquia de exceções,

---

<sup>11</sup> Tecnicamente, a exceção não foi lançada no código contido no método *try*, mas sim no método *Parse* que é invocado nesse código.

de onde emanam dois ramos, um para as excepções geradas pelo código da plataforma .NET, e outro para as excepções geradas pelo código das aplicações (veremos mais adiante como criar as nossas próprias excepções). O primeiro ramo é representado pela classe/excepção *SystemException*, e o segundo, pela classe *ApplicationException*.<sup>12</sup>

O facto de existirem diferentes tipos de excepções permite-nos ser selectivos nas excepções que queremos apanhar em blocos *catch*. Podemos assim lidar de forma diferente com diferentes excepções. Pode ser que em relação a algumas possamos contornar o erro, mas que em relação a outras nada possamos fazer, devendo deixar que a excepção seja apanhada noutra parte do código, ou mesmo que leve à terminação da aplicação.

A forma de indicar num bloco *catch* quais as excepções que queremos aí apanhar passa por identificar um tipo de excepção, indicando o nome da sua classe. Serão apanhadas não só as excepções dessa classe, mas também as das classes que dela derivam. Por exemplo, ao seleccionar a classe *SystemException* estamos a seleccionar todas as classes geradas por código da plataforma .NET. Exemplo:

```
static void Main(string[] args)
{
    if (args.Length != 2)
        Console.WriteLine("Utilização: ex1.exe <num1> <num2>");
    else
    {
        try
        {
            int a = int.Parse(args[0]);
            int b = int.Parse(args[1]);
            int soma = a + b;
            Console.WriteLine("{0} + {1} = {2}", a, b, soma);
        }
        catch (FormatException) // apanhar as excepções deste tipo
        {
            Console.WriteLine("Erro: Pelo menos um dos parâmetros não é um número.");
        }
    }
}
```

Opcionalmente, podemos indicar também o nome de uma variável, sendo que nesse caso, poderemos aceder ao objecto correspondente à excepção. No exemplo seguinte, vamos escrever na consola a mensagem de erro incluída no objecto que corresponde à excepção.

---

<sup>12</sup> Por convenção, os nomes das classes/excepções terminam todas com a palavra *Exception*.

```

static void Main(string[] args)
{
    if (args.Length != 2)
        Console.WriteLine("Utilização: ex1.exe <num1> <num2>");
    else
    {
        try
        {
            int a = int.Parse(args[0]);
            int b = int.Parse(args[1]);
            int soma = a + b;
            Console.WriteLine("{0} + {1} = {2}", a, b, soma);
        }
        catch (FormatException e)
        {
            Console.WriteLine("Erro: {0}.", e.Message);
        }
    }
}

```

Suponhamos que o utilizador invoca o programa da seguinte forma:

```
ex1.exe 255 abc
```

Uma vez que o segundo parâmetro não é um número, vai ocorrer um erro, sendo escrita a seguinte mensagem na consola:

```
Erro: Input string was not in a correct format..
```

O texto que aparece a seguir a “Erro:” foi obtido através da propriedade *Message* do objecto correspondente à excepção, que é acedido através da variável *e*.

Podemos escrever vários blocos *catch* a seguir a um bloco *try*, indicando em cada um dos blocos o tipo de excepções a apanhar. Desta forma, podemos lidar com diferentes excepções de forma diferente.

No exemplo apresentado a seguir temos dois blocos *catch*. O primeiro bloco apanha apenas as exceções do tipo *FormatException*. O segundo bloco apanha exceções de todos os tipos, uma vez que indica a classe *Exception*, que está na base da hierarquia de exceções (todas as classes exceção derivam dela).

```
static void Main(string[] args)
{
    try
    {
        int a = int.Parse(args[0]);

        int b = int.Parse(args[1]);

        int soma = a + b;

        Console.WriteLine("{0} + {1} = {2}", a, b, soma);
    }

    catch (FormatException)
    {
        Console.WriteLine("Erro: Não indicou dois números.");
    }

    catch (Exception e)
    {
        Console.WriteLine("Erro: {0}.", e.Message);
    }
}
```

Suponhamos que o utilizador invoca o programa da seguinte forma:

ex1.exe 255

Uma vez que só foi passado um parâmetro para a aplicação, vai ocorrer um erro diferente, e a exceção correspondente vai ser apanhada apenas no segundo bloco *catch*, fazendo com que seja escrita a seguinte mensagem na consola:

Erro: Index was outside the bounds of the array..

Se não tivéssemos incluído o segundo bloco *catch*, a aplicação iria crashar, uma vez que a exceção não seria apanhada. Lendo a mensagem de erro, podemos deduzir que a exceção foi lançada quando o programa tentou aceder ao segundo parâmetro através do array *args*.

A classe *Exception*

A classe *Exception* implementa funcionalidades que são herdadas pelas restantes classes que representam exceções. As propriedades mais importantes da classe são as seguintes:

- *Message* – Propriedade de leitura do tipo *string* que contém uma descrição da razão que levou ao lançamento da exceção.
- *InnerException* – É uma referência para uma possível exceção anterior que tenha levado ao lançamento da exceção actual. Pode ser nula, indicando que esta exceção não foi lançada num bloco *catch* de outra exceção.
- *StackTrace* – Propriedade de leitura do tipo *string* que indica a sequência de chamadas (*stack trace*) na altura em que a exceção foi lançada.

O método *ToString* da classe *Exception* retorna uma *string* que descreve a exceção. Exemplo:

Unhandled Exception: System.FormatException: Input string was not in a correct format.

```
at System.Number.StringToNumber(String str, NumberStyles options, NumberBuffer& number,
NumberFormatInfo info, Boolean parseDecimal)
```

```
at System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
```

```
at ex1.Program.Main(String[] args) in C:\Documents and Settings\Administrador\Os meus documentos\Visual
Studio 2005\Projects\chapter-exceptions\ex1\Program.cs:line 15
```

A classe *Exception* possui vários construtores. As assinaturas dos três construtores mais importantes são indicadas a seguir:

- *Exception()*
- *Exception(string message)*
- *Exception(string message, Exception innerException)*

Estes três construtores permitem inicializar o estado da exceção com mais ou menos informação.

## Criação de novos tipos de exceções

É natural que quando se cria uma aplicação se queira definir novos tipos de exceções, que vão ser lançadas quando ocorrerem erros específicos da aplicação. Para definir um novo tipo de exceção, temos que criar uma

nova classe que derive da classe *ApplicationException*, ou de outra classe dela derivada, podendo construir a nossa própria sub-hierarquia de exceções.

No exemplo seguinte vamos criar a exceção *NotANumber*, derivada directamente da classe *ApplicationException*.

```
class NotANumberException : ApplicationException
{
    NotANumberException(string value, Exception innerException)
        : base("O valor " + value + " não é um número", innerException)
    {
    }
}
```

Esta classe só possui um construtor, recebendo como parâmetros o valor que não pôde ser convertido para um número, e a exceção que a tentativa de conversão causou. Caso não queiramos indicar a exceção original que levou ao lançamento da exceção *NotANumberException*, podemos passar o valor *null* no segundo parâmetro do construtor.

Esta nova exceção poderia ser utilizada da seguinte forma:

```
try
{
    string valor = Console.ReadLine();

    int i = int.Parse(valor);

    Console.WriteLine("int: {0}", i);
}

catch (Exception e)
{
    throw new NotANumberException(valor, e); // lançamento de exceção
}
```

### Libertação de recursos

Quando uma exceção é lançada, o fluxo normal de execução é interrompido, podendo fazer com que recursos alocados não sejam devidamente libertados. Para ajudar a evitar essa situação, o mecanismo de exceções permite incluir um bloco de código *finally* a seguir aos blocos *catch* (ou a seguir ao bloco *try*, se não houver



nenhum bloco *catch*). O código contido num bloco *finally* é executado independentemente de ser ou não lançada uma excepção dentro do bloco *try*. Assim sendo, se fizermos aí a libertação dos recursos, temos a garantia de que eles vão ser libertados logo que deixarem de ser necessário, independentemente da ocorrência de erros inesperados.

Segue-se um exemplo que inclui um bloco *finally*:

```
try
{
    baseDados.Open();

    Processar(baseDados);
}
finally
{
    if (baseDados.IsOpen) baseDados.Close();
}
```

### Lançamento de excepções

O lançamento de uma excepção deve ser feito quando ocorre um erro inesperado. Para isso utiliza-se o operador *throw*, seguido do objecto excepção, que é construído utilizando um dos construtores disponibilizados na respectiva classe. Segue-se um exemplo em que é lançada uma excepção do tipo *NotANumberException* (definido anteriormente):

```
throw new NotANumberException(a, e);
```

## Exercícios

1. Em qual dos blocos *try*, *catch* ou *finally* é incluído o código que só será executado se for lançada uma excepção? Escolha a opção correcta:
2. Em qual dos blocos *try*, *catch* ou *finally* é normalmente incluído o código que pode gerar excepções, e deve ser protegido? Escolha a opção correcta:
3. O que é escrito na consola quando é executado o código apresentado a seguir?

```
try {  
    Console.Write("1");  
    throw new Exception("Erro");  
    Console.Write("2");  
} catch {  
    Console.Write("3");  
} finally {  
    Console.Write("4");  
}
```

Escolha a opção correcta:

- ☐ 1234
  - ☐ 123
  - ☐ 124
  - ☐ 134
  - ☐ 34
4. Crie uma aplicação de consola que receba dois números como parâmetros e escreva o resultado da sua divisão. Utilize o mecanismo de excepções da linguagem C# para evitar que a aplicação termine de forma abrupta quando o segundo parâmetro for zero.

5. Crie um projecto de consola com as classes indicadas a seguir.

A classe *NotANumberException* vai ser utilizada para lançar uma excepção no método *Processar*. Esse método recebe uma *string* e tenta convertê-la para um *int* com o método *int.Parse*. Se falhar, tenta convertê-la para um *double*. Se falhar novamente, lança a excepção *NotANumberException*. **Nota:** aproveite o facto de o método *Parse* lançar uma excepção quando a conversão falha.

O método *Main* deve conter o seguinte código:

```
try
{
    Processar("1");
    Processar("1,5");
    Processar("-15");
    Processar("1a");
    Processar("teste");
}
catch (NotANumberException e)
{
    Console.WriteLine(e.Message);
}
finally
{
    Console.WriteLine("Terminado");
}
```

A execução da aplicação deve produzir o seguinte resultado:

```
Processar: 1
int: 1
Processar: 1,5
double: 1,5
Processar: -15
int: -15
Processar: 1a
A string 1a não representa um número
Terminado
```

## Referências

- <http://www.softsteel.co.uk/tutorials/cSharp/>
- <http://www.csharp-station.com/Tutorial.aspx>
- <http://csharpcomputing.com/Tutorials/TOC.htm>
- [http://msdn2.microsoft.com/en-us/library/9b9dty7d\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/9b9dty7d(VS.80).aspx)
- [http://msdn2.microsoft.com/en-us/library/2s05feca\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/2s05feca(VS.80).aspx)
- C# School – Programmers Heaven (cf. Site da Disciplina)

## Parte IX – Windows Forms

Autor: João Carlos Silva

---

Esta parte do documento C# Essencial, dá início ao estudo da biblioteca *Windows Forms* a qual permite desenvolver aplicações Windows. A biblioteca está definida no *namespace System.Windows.Forms*.

### Manipulação de Forms

Através da biblioteca *Windows Forms* podemos desenvolver vários tipos de interfaces Windows, por exemplo:

MDI (Multiple Document Interface): Aplicação que suporta múltiplos documentos abertos simultaneamente (ex: Word)

SDI (Single Document Interface): Aplicação que permite a abertura de apenas um documento de cada vez (ex: NotePad)

Janelas modais: Janelas informativas conhecidas como diálogos.

### Interface MDI

#### Definição e inserção de um menu de opções

De modo a criar uma aplicação Windows com a plataforma Microsoft Visual Studio .NET, é necessário criar um novo projecto com as opções *File → New Project*, seleccionar o tipo de projecto *Visual C# Projects*, e finalmente o template *Windows Application*. Por defeito é gerado automaticamente um formulário do tipo SDI com nome *Form1*.

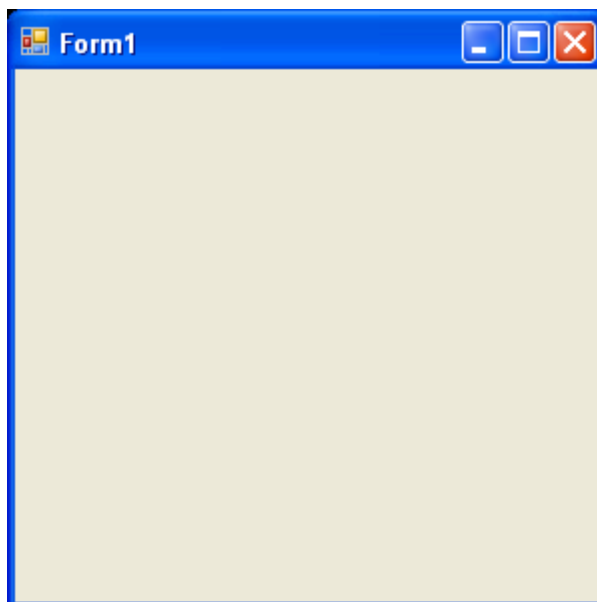


Figura 17 – Formulário SDI

Para proceder a definição de uma interface MDI devemos alterar a propriedade do formulário principal *IsMdiContainer* para *true*.

A inserção de um menu na janela principal passa por activar a *toolbox* e de seguida arrastar para a janela principal o componente *MenuStrip*. Finalmente procede-se a definição e inserção dos respectivos itens de menu.

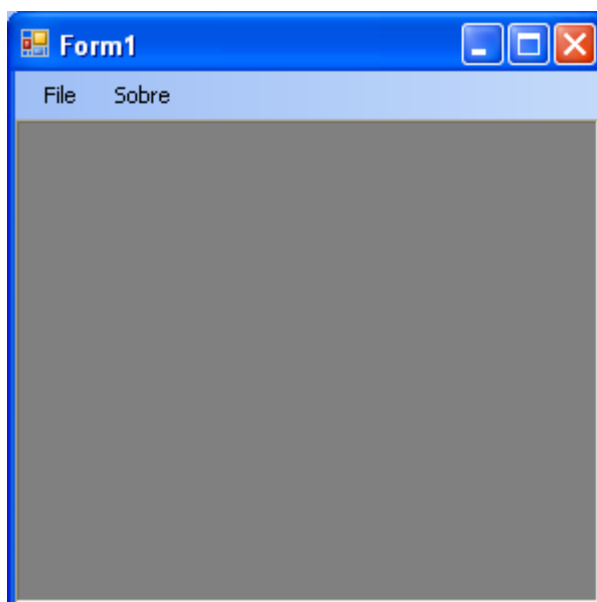


Figura 18 - Formulário MDI

Para cada item de menu é possível associar código ao evento click do *MenuItem*, por exemplo:

```
private void menuItem1_Click(object sender, System.EventArgs e)

{ MessageBox.Show("Olá Mundo!");

}
```

Este código referencia o método estático *Show* da classe *MessageBox* com o objectivo de mostrar uma mensagem ao utilizador da aplicação. A instrução permite abrir uma nova janela com o texto passado por parâmetro e um botão *Ok* para fechar.

```
private void menuItem4_Click(object sender, System.EventArgs e)

{ Application.Exit( );

}
```

Neste caso, faz-se uso da classe *Application* e o seu método estático *Exit( )* para fechar a aplicação.

Existem ainda muitas propriedades que podem ser configuradas para cada formulário. Como por exemplo a propriedades *ShowInTaskBar* e *WindowState*. A primeira faz com que a aplicação seja mostrada, ou não, na barra de tarefas. A segunda permite-nos definir o estado inicial do formulário: *Minimized*, *Normal* ou *Maximized*.

Para adicionar um novo formulário (do tipo SDI) à aplicação devemos executar as seguintes opções do menu *Project* → *Add New Item* e seleccionar *Windows Forms*. Por defeito o formulário tem nome *Form2*.

Neste momento o projecto é composto por um formulário (*Form1*) do tipo MDI e outro (*Form2*) do tipo SDI.

### Adicionar uma nova janela filha

Vamos agora implementar uma opção no menu da janela pai (formulário *Form1*). A execução da opção deverá inserir uma nova janela filha. O código deverá ser o seguinte:

```
Form2 form = new Form2( );
form.MdiParent = this;

form.Text = "Janela Filha";
form.Show( );
```

O primeiro passo consiste em criar uma nova instância do formulário que constitui a nossa janela filha (neste caso *Form2* do tipo SDI).

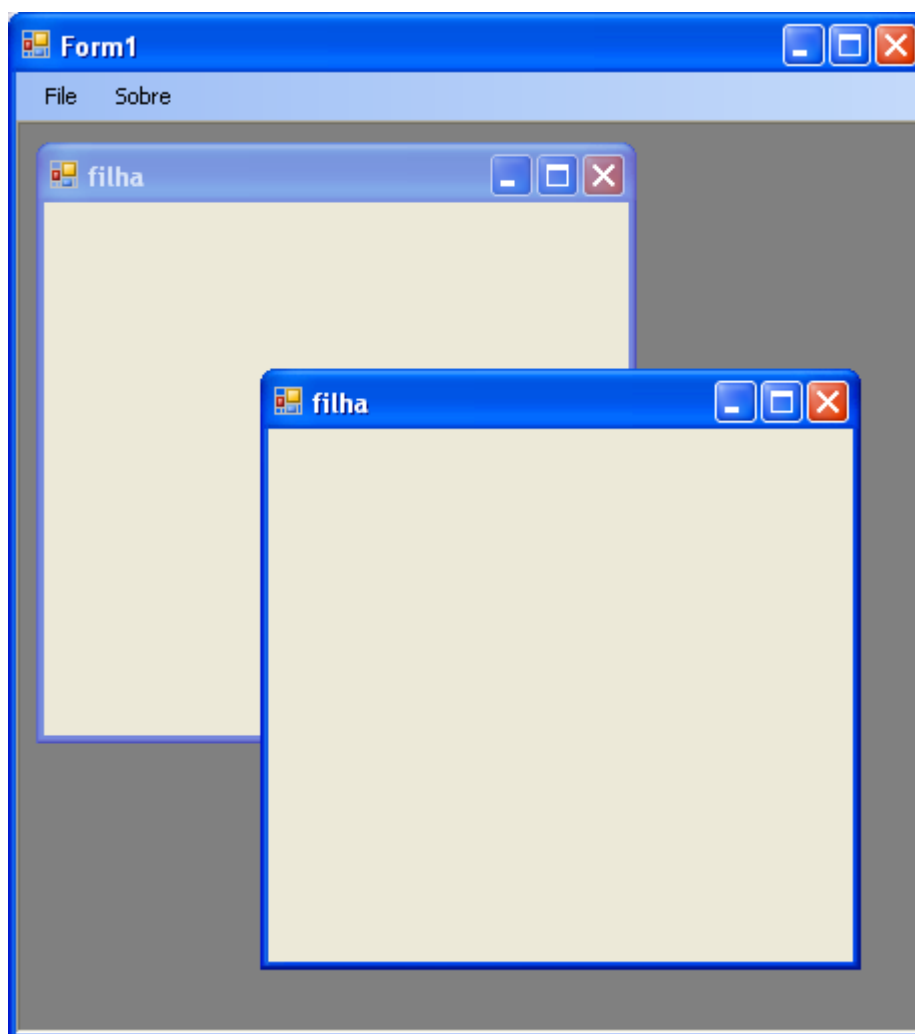


Figura 19 - Formulário MDI com dois formulários SDI

A seguir, definimos o pai da janela filha alterando a propriedade *MdiParent* do formulário como sendo o formulário principal. A atribuição do valor *this* a essa propriedade permite indicar que a janela pai é o objecto correspondente ao formulário principal. Para terminar devemos ainda mostrar a janela filha invocando o método *Show()*.

### Fechar a janela filha activa

Para fechar a janela filha activa é necessário definir uma nova opção no menu da janela pai, por exemplo a opção “Fechar janela activa”. A opção deverá estar associada ao código seguinte:

```
this.ActiveMdiChild.Close();
```

A propriedade *ActiveMdiChild* é um objecto do tipo *Form*, mais precisamente o objecto da janela filha activa nesse instante. A execução do método *Close()* permite encerrar de imediato a janela activa.



## Organização das janelas filhas

As janelas filhas podem ser organizadas em cascata, lado a lado na horizontal e na vertical. Para o efeito basta executar na janela pai respectivamente o código seguinte:

```
this.LayoutMdi(System.Windows.Forms.MdiLayout.Cascade); // em cascata
```

```
this.LayoutMdi(System.Windows.Forms.MdiLayout.TileHorizontal); // na horizontal
```

```
this.LayoutMdi(System.Windows.Forms.MdiLayout.TileVertical); // na vertical
```

## Utilização do componente TabControl

O componente *TabControl* permite agrupar vários formulários num só. A sua inserção é realizada através da *ToolBox* arrastando para a janela o componente *TabControl*. A seguir clicando no botão direito do rato temos acesso as opções *Add Tab* e *Remove Tab* as quais permitem adicionar ou remover páginas de um *TabControl*. Para ter acesso às diversas páginas de um *TabControl*, basta clicar na propriedade *TabPage* de modo a visualizar uma janela com as páginas criadas.



Figura 20 – Componente TabControl

## O componente *ErrorProvider*

Para validar os dados inseridos nos componentes pelo utilizador podemos utilizar o componente *ErrorProvider*. Por exemplo para validar o conteúdo de uma *TextBox* (`TextBox tb = new TextBox();`) utilizando o componente *ErrorProvider* devemos primeiro alterar a propriedade *CausesValidation* da *TextBox* para *true*.

```
if (tb.Text.Length < 6) errorProvider.SetError(tb, "Código inválido");
```

Este código permite lançar a mensagem “Código inválido” sempre que o conteúdo inserido na *TextBox* tenha menos de 6 caracteres.

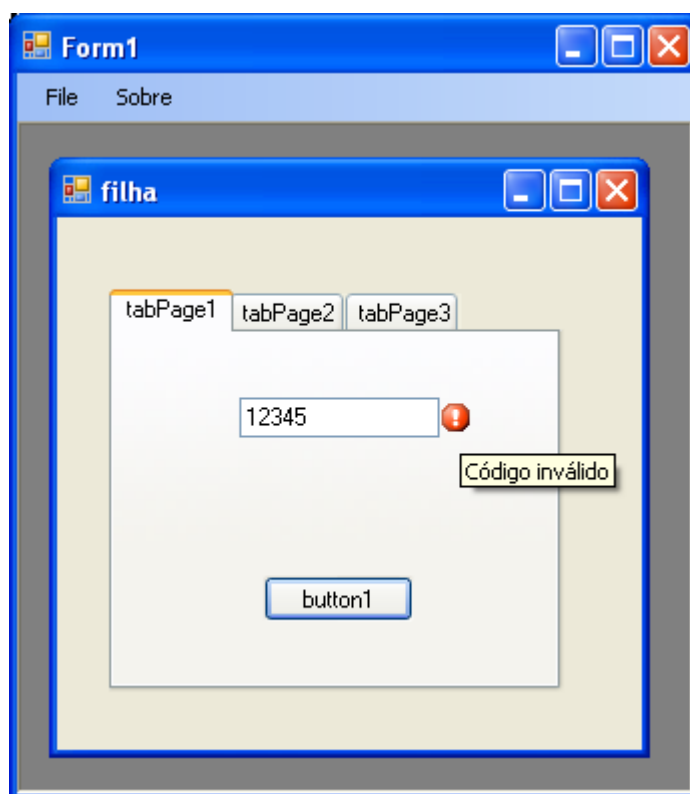


Figura 21 – Componente *ErrorProvider*

## Desenvolvimento de um editor de texto

No desenvolvimento de um editor de texto podemos aplicar o componente *RichTextBox*. Este componente permite editar, copiar, colar, localizar e formatar texto.

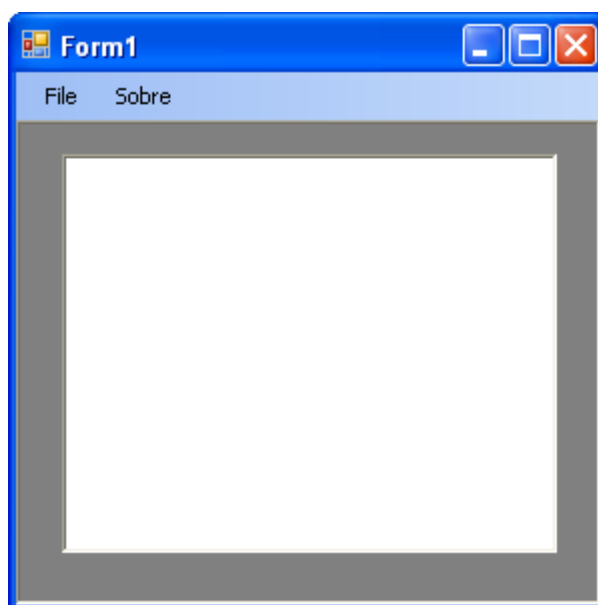


Figura 22 – Componente RichTextBox

```
RichTextBox richEditor = new RichTextBox();
```

O método para copiar o texto seleccionado:

```
private void copiar_Click(object sender, System.EventArgs e)
{ if (richEditor.SelectedText != "") richEditor.Copy( );
}
```

O método para colar o texto seleccionado:

```
private void colar_Click(object sender, System.EventArgs e)
{ richEditor.Paste( );
}
```

Para ler e guardar um ficheiro no sistema podemos executar os componentes OpenFileDialog e SaveFileDialog:

```
private void abrir_Click(object sender, System.EventArgs e)
{ of = new OpenFileDialog( );
  of.ShowDialog( );
  ...
  if (of.ShowDialog( ) == DialogResult.OK)
      this.richEditor.LoadFile(of.FileName, RichTextBoxStreamType.RichText);
}
```

Para determinar qual foi o botão seleccionado pelo utilizador no diálogo devemos utilizar a lista *DialogResult*:

Ok: O botão clicado no diálogo foi *OK*

Cancel: O botão clicado no diálogo foi *Cancel*

Yes: O botão clicado no diálogo foi *Yes*

No: O botão clicado no diálogo foi *No*

Ignore: O botão clicado no diálogo foi *Ignore*

Retry: O botão clicado no diálogo foi *Retry*

None: Nenhum botão foi clicado

Abort: O botão clicado no diálogo foi *Abort*

Para abrir o ficheiro seleccionado:

```
this.richEditor.LoadFile(of.FileName,
    System.Windows.Forms.RichTextBoxStreamType.RichText);
```

Para guardar um ficheiro:

```
sf = new SaveFileDialog( );

if (sf.ShowDialog( ) == DialogResult.OK)

    this.richEditor.SaveFile(sf.FileName, RichTextBoxStreamType.RichText);
```

Para imprimir o ficheiro devemos utilizar os componentes *PrintPreviewDialog*, *PrintDialog* e *PrintDocument*.

O componente *PrintPreviewDialog* deve ser invocado para pre-visualizar um documento:

```
private void printPreview_Click(object sender, System.EventArgs e)

{
    stringR = new StringReader(this.richEditor.Text);

    PrintPreviewDialog printPreviewDialog1 = new PrintPreviewDialog( );

    printPreviewDialog1.Document = this.printDocument1 ;

    printPreviewDialog1.ShowDialog( );
}
```

O componente *PrintDialog* deve ser invocado para mostrar o diálogo para o envio de um documento para a impressora:

```
private void imprimir_Click(object sender, System.EventArgs e)

{
    stringR = new

    StringReader(this.richEditor.Text);
```

```
printDialog1.Document = this.printDocument1;  
  
if (printDialog1.ShowDialog() == DialogResult.OK) this.printDocument1.Print();  
  
}
```

## Os componentes indispensáveis

Crie um novo projecto no visual studio com template Windows Application, e manipule os componentes seguintes.

Para cada componente consulte todas a propriedades disponíveis:

- Button
- Label
- TextBox
- ComboBox
- ListBox
- Checked ListBox
- Radio Button
- TreeView
- ListView
- PictureBox
- DateTimePicker
- MonthCalendar

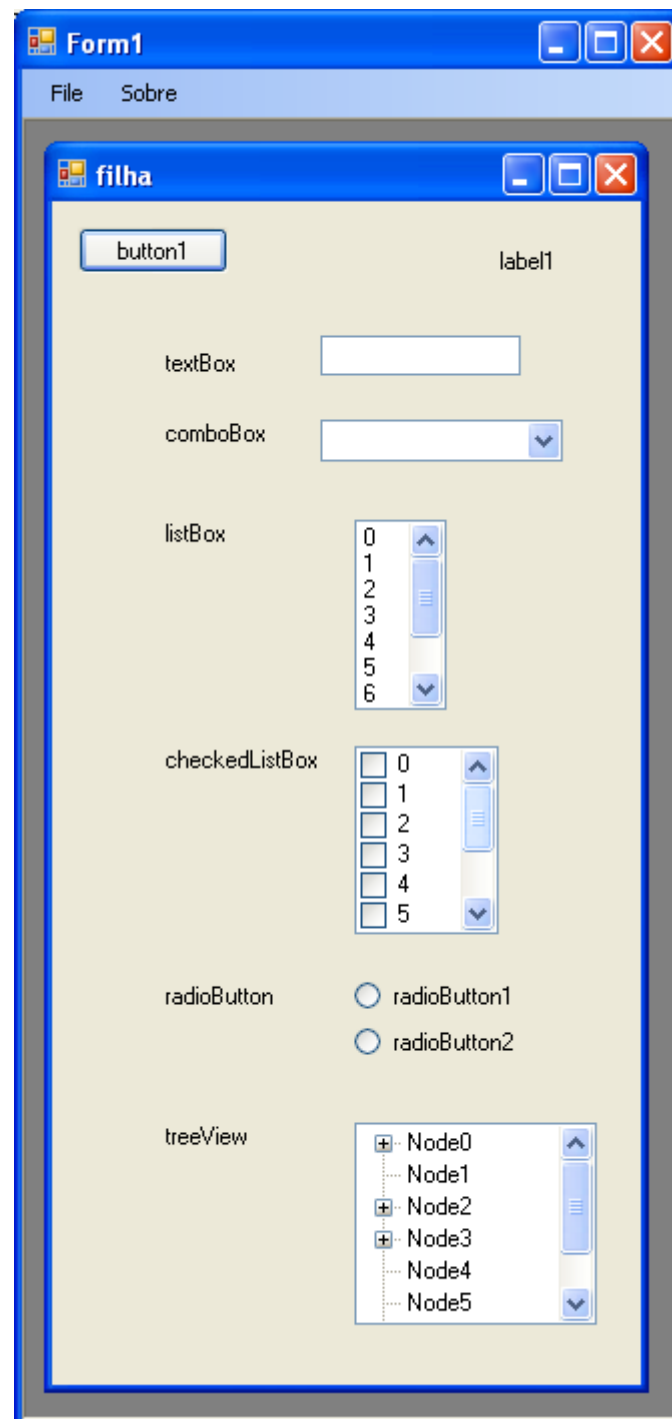


Figura 23 - Componentes Button, Label, TextBox, ComboBox, Checked ListBox, Radio Button e TreeView



Figura 24 - Componentes PictureBoxDate, TimePicker e MonthCalendar

## Exercícios

1. Pretende-se implementar uma aplicação Windows através da biblioteca *Windows Forms*. A aplicação deverá implementar uma agenda de contactos telefónicos.

Cada contacto é definido com o *nome* (os nomes são únicos) e o número de *telemóvel*;

As funcionalidades a disponibilizar na interface gráfica são:

- número total de contactos
- adicionar um novo contacto na agenda
- remover um contacto dado o nome
- determinar a existência de um contacto dado o nome
- obter o número de telemóvel dado o nome do contacto
- listar os nomes de todos os contactos por ordem crescente
- guardar em ficheiro
- ler o ficheiro de dados
- Pre-visualizar e imprimir

Define uma interface MDI com múltiplas janelas filha. Organize as janelas lado a lado e define um ícone na barra *Tray*. Deverá utilizar os componentes *TabControl*, *ComboBox*, *Check ListBox*, *Radio Button*, *TreeView* e *ListView*.

2. Define uma aplicação Windows para gestão de um stand automóvel. Para cada automóvel o sistema deverá armazenar os dados seguinte:
  - matrícula
  - marca
  - modelo
  - cilindrada
  - quilómetros
  - preço
  - estado (a venda ou vendido)

Cada cliente do stand deverá ficar armazenado no sistema com a informação seguinte:

- número de bilhete de identidade
- nome
- morada
- lista das matrículas dos veículos comprados no stand

Define uma interface MDI com múltiplas janelas filha. Organize as janelas lado a lado. Deverá utilizar os componentes *TabControl*, *ComboBox*, *Check ListBox*, *Radio Button*, *TreeView* e *ListView*.

## Referências

1. <http://www.softsteel.co.uk/tutorials/cSharp/>



2. <http://www.csharp-station.com/Tutorial.aspx>
3. <http://csharpcomputing.com/Tutorials/TOC.htm>
4. [http://msdn2.microsoft.com/en-us/library/9b9dty7d\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/9b9dty7d(VS.80).aspx)
5. [http://msdn2.microsoft.com/en-us/library/2s05feca\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/2s05feca(VS.80).aspx)
6. C# School – Programmers Heaven (cf. Site da Disciplina)

## Parte X – Manipulação de Ficheiros e Serialização

Autor: Luís Ferreira

---

Esta parte do documento C# Essencial dá início ao tratamento de Ficheiros e Serialização

### Considerações

Quando se fala em manipular ficheiros entramos no contexto do chamado “acesso a dados” que, na terminologia .NET poderá abarcar essencialmente Ficheiros e Serialização, XML e Bases de Dados.

Em C# a manipulação de ficheiros não é directa (como acontece em C, Pascal e outras linguagens), ie, operações de escrita e leitura (I/O) são realizadas através de *streams*.

O NameSpace a utilizar para manipular ficheiros é *System.IO*.

### Patterns para manipulação de Ficheiros

Manipular um ficheiro implica conseguir criá-lo, definir o tipo de conteúdo que vai possuir e o tipo de acesso que se pretende. Um ficheiro surge em formato texto ou formato binário (embora efectivamente todos os ficheiros sejam guardados em formato binário!). Na prática, num ficheiro de texto, caso se pretenda guardar o valor 42, é guardado o ASCII equivalente, ie, 0x34 e 0x32. No formato binário é usado o valor 42. Qualquer editor de texto consegue apresentar em pleno um ficheiro de texto.

### Ficheiros de Texto

Path do ficheiro

- *Indicar o nome e local (a path ) do ficheiro:*

```
string Filename = @"C:\Documents and Settings\Employees.spr";
```

ou

```
string Filename = "C:\\Documents and Settings\\Employees.spr";
```

- *Directoria actual*

```
//actual directoria de trabalho
```

```
Directory.GetCurrentDirectory();
```

```
//directoriar onde a aplicação está a executar
```

```
Application.StartupPath
```

```
//definir nova directoria actual
```

```
Directory.SetCurrentDirectory( Application.StartupPath );
```

Criar um ficheiro de texto para escrita: Classe *StreamWriter* e método *File.CreateText()*

```
private void SaveText(string text, string file)
{
    StreamWriter sw = null;
    try
    {
        sw = File.CreateText(file);
        sw.Write(text);
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);
    }
    finally
    {
        if (sw != null)
            sw.Close();
    }
}
```

Exemplo de invocação:

```
SaveText("Isto é um teste","aula.txt");
```

Outra forma de escrever:

```
StreamWriter sw = File.CreateText("aula.txt");
sw.WriteLine("Hello file");
```

Outra forma:

```
TextWriter tw = new StreamWriter("teste.txt");
```

Abrir o ficheiro em modo *append*:

```
StreamWriter asw = new StreamWriter("teste.txt", true);
```

#### **Nota:**

A classe *StreamWriter* é adequada para gravar em ficheiros de texto. Caso se pretenda gravar em ficheiros HTML existe a classe *HtmlTextWriter*.

Abrir um ficheiro para leitura: Classe *StreamReader* e método *File.OpenText()*

```

public static string ReadFromFile(string filename)
{
    string s = "", aux = "";
    try
    {
        using (StreamReader sr = File.OpenText(filename))
        {
            while ((aux = sr.ReadLine()) != null)
            {
                s += aux + "\r\n";
            }
            sr.Close();
        }
    }
    catch (Exception e)
    {
        throw new FileNotFoundException("Problema ..." + e.Message);
    }
    return s;
}

```

Outra forma de associar um stream a um ficheiro consegue-se com a classe *FileStream*:

```

FileStream fs = new FileStream(fileName, FileMode.Open);
using (StreamReader sr = new StreamReader(fs))
{
    //...
}

```

O exemplo seguinte mostra uma possível implementação do comum comando DOS “Type”:

```

/// <summary>
/// Mostra conteúdo do ficheiro
/// </summary>
/// <param name="fileName">Nome do ficheiro</param>
public static void Type(string fileName)
{
    try
    {
        using (StreamReader sr = new StreamReader(fileName))
        {
            String line;
            while ((line = sr.ReadLine()) != null)
            {
                Console.WriteLine(line);
            }
        }
    }
    catch (Exception e)
    {
        Console.WriteLine("Ficheiro desconhecido.." + e.Message);
    }
}

```

## Ficheiro Binários

Criar um ficheiro binário para escrita: Classe *FileStream* e método *File.Create()*

```
private void SaveBinary(byte[] bytes, string file)
{
    FileStream fs = null;
    try
    {
        if (File.Exists(file))
        {
            File.Delete(file);
        }
        fs = File.Create(file);
        fs.Write(bytes, 0, bytes.Length);
    }
    catch (Exception e)
    {
        MessageBox.Show(e.Message);
    }
    finally
    {
        if (fs != null)
            fs.Close();
    }
}
```

Existem outras formas de conseguir o mesmo. A classe *FileStream* permite definir de forma explícita o tipo de acesso que se permite ter ao ficheiro. Veja-se o seguinte exemplo:

```

public static void SaveToFile(string fileName, string t)
{
    if (File.Exists(fileName))           //se existe, append
    {
        using (StreamWriter sw = new StreamWriter(fileName))
        {
            sw.WriteLine(t);
            sw.Close();
        }
    }
    else                                   //senão, cria e grava
    {
        try
        {
            FileStream fs = new FileStream(fileName, FileMode.Append);
            using (StreamWriter sw = new StreamWriter(fs))
            {
                sw.WriteLine(t);
                sw.Close();
            }
        }
        catch (FileNotFoundException e)
        {
            //return erro!
            throw new FileNotFoundException("Problema..." + e.Message);
        }
    }
}

```

Existem várias formas de criar instâncias da classe `FileStream`:

```

//abre ficheiro para escrita
FileStream fs = File.OpenWrite(fileName);

//abre ficheiro para leitura
FileStream fs = File.OpenRead(fileName);

//cria ficheiro consoante o modo pretendido
FileStream fs = new FileStream(fileName, FileMode, FileAccess, FileShare);

```

Onde os enumerados possíveis são definidos por:

FileMode	Descrição
Create	<i>FileMode.Create</i> cria um ficheiro e se existir faz override
CreateNew	Cria um ficheiro e se existe gera uma exceção <i>IOException</i> .
Append	Abre um ficheiro e prepara para escrever no final. Se o ficheiro não existe, cria-o. Só pode usar <i>FileMode.Append</i> com <i>FileAccess.Write</i> senão é gerada a exceção <i>ArgumentException</i> .
Open	Abre um ficheiro. Se não existir gera a exceção <i>FileNotFoundException</i> .

<b>OpenOrCreate</b>	Abre um ficheiro. Se não existir, cria-o.
<b>Truncate</b>	Abre um ficheiro e “limpa-o”.

FileAccess	Descrição
<b>Read</b>	Define acesso para leitura
<b>Write</b>	Define acesso para escrita
<b>ReadWrite</b>	Define acesso para escrita/leitura

O Enumerado *FileShare* sai do âmbito deste documento

**Verificar se um ficheiro existe:** Independentemente do formato do ficheiro, verificar se o ficheiro existe ou não é conseguido com o método *File.Exists()*.

Como já vimos no exemplo anterior:

```
if (File.Exists("C:\\aula.txt"))
{
    .....
}
```

### Outros métodos disponíveis em *File.IO*

<i>File.Exists(filename)</i>	<i>true</i> se existir
<i>File.Delete(filename)</i>	Apaga o ficheiro
<i>File.AppendText(String)</i>	Acrescenta texto ao final do ficheiro
<i>File.Copy(fromFile, toFile)</i>	Copia o ficheiro
<i>File.Move(fromTile, toFile)</i>	Move o ficheiro, apagando o original
<i>File.GetExtension(filename)</i>	Devolve a extensão do ficheiro
<i>File.HasExtension(filename)</i>	True se o ficheiro tem extensão

As classes *FileInfo*, *Directory* e *DirectoryInfo* possuem vários objectos para manipular as propriedades de directorias e ficheiros.

## Manipular ficheiros em Windows Forms

Em Windos Forms, para manipular directorias e ficheiros os objectos mais comuns são o *OpenFileDialog* e *FolderBrowserDialog*.

Quando se utiliza o *OpenFileDialog* é necessário definir o filtro (extensões de ficheiros), no sentido de filtrar os ficheiros que se pretendem. A sintaxe é:

[Texto] [Lista de extensões separadas por “;”]

Exemplo:

```
openFileDialog1.Filter = "Word (*.doc) | *.doc;*.rtf|Text (*.txt) | *.txt|All (*.*) | *.*";
```

No exemplo seguinte, uma textbox é preenchida com o nome do ficheiro seleccionado.

```
private void button1_Click(object sender, EventArgs e)
{
    OpenFileDialog dlg = new OpenFileDialog();
    dlg.Title = "Ficheiro a ler...";
    dlg.DefaultExt = ".txt";
    dlg.InitialDirectory=@"c:\temp";
    dlg.Filter = "Word (*.doc) | *.doc;*.rtf|Text (*.txt) | *.txt|All (*.*) | *.*";
    dlg.FilterIndex = 2 ;
    dlg.RestoreDirectory = true;
    dlg.FileName = "";
    if (dlg.ShowDialog() == DialogResult.OK)
    {
        textBox1.Text = dlg.FileName;
    }
}
```



Figura 25 – Botão "Browse"



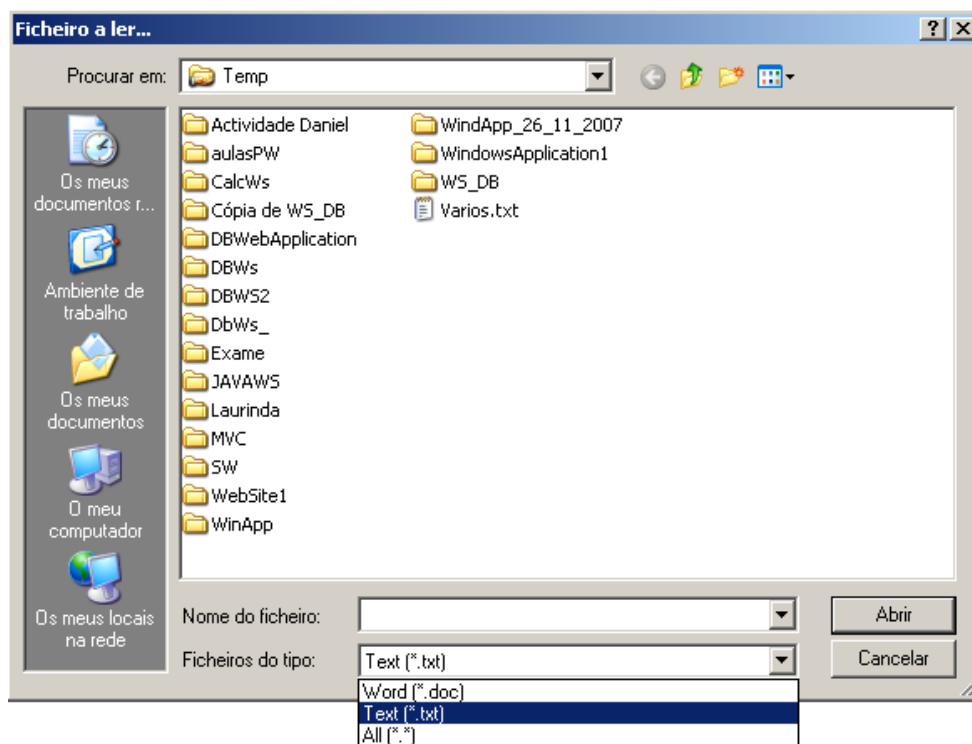


Figura 26 – Controlo OpenFileDialog

Para manipular directorias utiliza-se o controlo *FolderBrowserDialog*. Veja-se o exemplo seguinte:

```
private void button2_Click(object sender, EventArgs e)
{
    FolderBrowserDialog fb = new FolderBrowserDialog();
    if (fb.ShowDialog() == DialogResult.OK)
    {
        MessageBox.Show(fb.SelectedPath);
    }
}
```

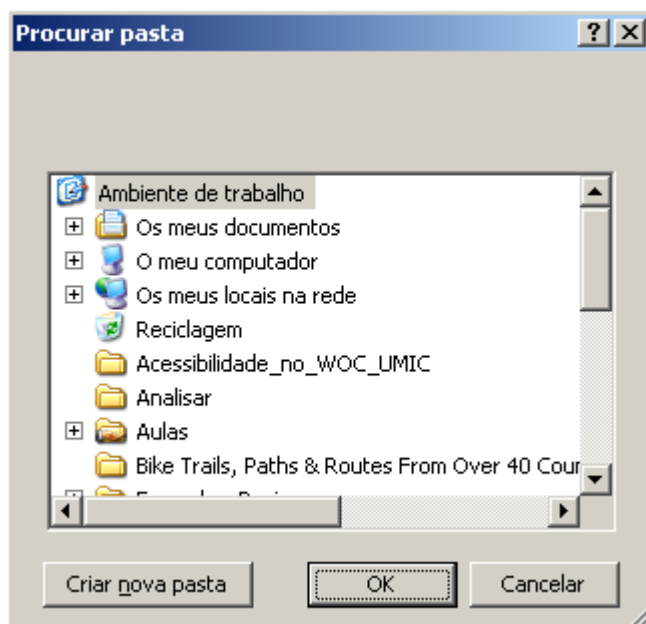


Figura 27 – Controlo FolderBrowserDialog

## Exemplos

*Método que lê um ficheiro em modo binário*

```

/// <summary>
/// Ler de um ficheiro em modo binário
/// </summary>
/// <param name="filename">Nome do Ficheiro</param>
/// <returns>Conteúdo do ficheiro</returns>
public static string ReadFromFileB(string fileName)
{
    string s = "";
    try
    {
        FileStream fs = new FileStream(fileName, FileMode.Open);
        using (BinaryReader br = new BinaryReader(fs, Encoding.Unicode))
        {
            s=br.ReadString();
            br.Close();
        }
    }
    catch (Exception e)

```

```

{
    //return erro!
    throw new FileNotFoundException("Problema..." + e.Message);
}
return s;
}

```

*Método que grava em modo binário:*

```

/// <summary>
/// Grava num ficheiro em modo Binário
/// </summary>
/// <param name="fileName">Nome do ficheiro</param>
public static void SaveToFileB(string fileName, string t)
{
    if (File.Exists(fileName))    //se existe, append
    {
        FileStream fs = new FileStream(fileName, FileMode.Create);
        using (BinaryWriter sw = new BinaryWriter(fs, Encoding.Unicode))
        {
            sw.Write(t);
            sw.Close();
        }
    }
    else    //senão, cria e grava
    {
        try
        {
            FileStream fs = new FileStream(fileName, FileMode.Append);
            using (BinaryWriter sw = new BinaryWriter(fs, Encoding.Unicode))
            {
                sw.Write(t);
                sw.Close();
            }
        }
        catch (FileNotFoundException e)
        {

```

```

        //return erro!
        throw new FileNotFoundException("Problema..." + e.Message);
    }
}
}

```

### Guardar um conjunto de valores para ficheiro

A classe *Exercicio* possui um array de inteiros que se propõe guardar em ficheiro. O primeiro registo do ficheiro tem o número total de elementos guardados.

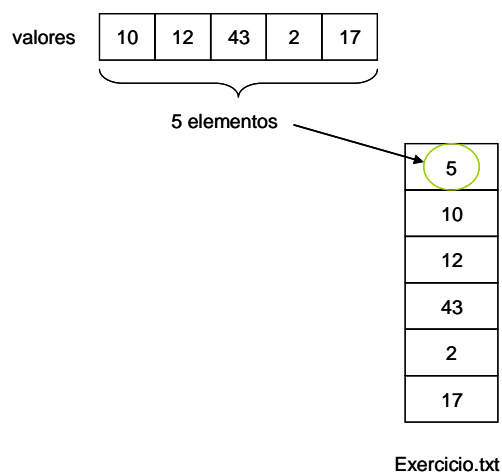


Figura 28 – Exercício

```

//IPCA-EST
using System;
using System.Collections;
using System.Text;
using System.IO;
namespace Aula_06_05_2008
{
    /// <summary>
    /// Guardar um conjunto de inteiros para ficheiro
    /// </summary>
    class Exercicio
    {
        //atributos
        public int[] valores;
    }
}

```

```

private int tot;
/// <summary>
/// construtor
/// </summary>
/// <param name="n"></param>
public Exercicio(int n)
{
    valores = new int[n];
    tot = n;
}

/// <summary>
/// Carrega vector com valores
/// </summary>
public void AddVals()
{
    for (int i = 0; i < tot; i++)
        valores[i] = i;
}

/// <summary>
/// Guarda todos os valores para ficheiro
/// Na 1ª posição do ficheiro guarda o total de elementos a gravar
/// </summary>
/// <param name="fileName">Nome do ficheiro</param>
public void SaveValores(string fileName)
{
    FileStream fs = new FileStream(fileName, FileMode.Create);
    BinaryWriter fb = new BinaryWriter(fs);

    if (valores.Length == 0)
        fb.Write(0);
    else
    {
        //primeira linha tem a dimensão do vector
        fb.Write(valores.Length);
        foreach (int v in valores)
            fb.Write(v);
    }
}

```

```

    }
    fb.Close();
}

/// <summary>
/// Carrega todos os valores existente en ficheiro
/// O primeiro valor corresponde ao número total de elementos          /// existentes no ficheiro
/// </summary>
/// <param name="fileName"></param>
public void LoadVals(string fileName)
{
    valores = null;
    FileStream fs = new FileStream(fileName, FileMode.Open);
    BinaryReader fb = new BinaryReader(fs);

    //ler a primeira linha=tamanho do vector
    tot=fb.ReadInt32();
    //tot = fb.Read();
    valores = new int[tot];
    for (int i = 0; i < tot; i++)
    {
        valores[i] = fb.ReadInt32();
        //valores[i] = fb.Read();
    }
    fb.Close();
}

/// <summary>
/// Método Auxiliar para mostrar o conteúdo do vector
/// </summary>
public void ShowVals()
{
    foreach (int v in valores)
        Console.WriteLine(v);
}
}

```

```
private void button1_Click(object sender, EventArgs e)
{
    Exercicio ex = new Exercicio(10);
    ex.AddVals();
    ex.SaveValores("Exercicio.txt");
    ex.LoadVals("Exercicio.txt");
    ex.ShowVals();
}
```

O exemplo seguinte mostra a aplicação de vários métodos da classe File.IO. (adaptado de <http://msdn.microsoft.com/en-us/library/system.io.file.aspx>)

```
class Test
{
    public static void Main()
    {
        string path = @"c:\temp\MyTest.txt";
        if (!File.Exists(path))
        {
            // cria ficheiro
            using (StreamWriter sw = File.CreateText(path))
            {
                sw.WriteLine("Viva");
                sw.WriteLine("o");
                sw.WriteLine("Benfica");
            }
        }

        // Abre o ficheiro para leitura
        using (StreamReader sr = File.OpenText(path))
        {
            string s = "";
            while ((s = sr.ReadLine()) != null)
            {
                Console.WriteLine(s);
            }
        }
    }
}
```

```
}

try
{
    string path2 = path + "temp";
    // garantir que o destino não existe
    File.Delete(path2);
    // copia o ficheiro
    File.Copy(path, path2);
    Console.WriteLine("{0} copiado para {1}.", path, path2);
    // apaga o ficheiro novo criado
    File.Delete(path2);
    Console.WriteLine("{0} foi apagado com sucesso.", path2);
}
catch (Exception e)
{
    Console.WriteLine("Algo correu mal: {0}", e.ToString());
}
}
```

## Referências

- C# School – Programmers Heaven (cf. Site da Disciplina)
- Microsoft® Visual C#® .NET 2003 Developer's Cookbook, ISBN : 0-672-32580-2
  - <http://www.functionx.com/vcsharp/fileprocessing/Lesson04.htm>
- O código fonte apresentado encontra-se disponível no WOC



## Parte XI – ADO.NET

Autor: Marco Costa

---

Esta parte do documento C# Essencial dá início ao estudo do acesso a bases de dados usando o *framework* ADO.NET.

### Sumário:

#### Introdução

O ADO.NET disponibiliza um conjunto de classes que permitem aceder a bases de dados.

A interacção com os Sistemas de Gestão de Bases de Dados (SGBD's) pode ser feita no modo tradicional ou no modo “desconectado”.

No modo tradicional o utilizador (programador) estabelece a ligação ao SGBD e interage com ele através de comandos SQL (ou *stored procedures*).

No modo desconectado o ADO.NET gere um *buffer* local, chamado *dataset*. A aplicação estabelece uma ligação automática ao SGBD quando é necessário buscar dados, e encerra-a logo de seguida, guardando os dados recebidos num *dataset*. O utilizador/programador pode realizar operações de actualização, inserção e remoção de dados no *dataset*. As alterações feitas ao *dataset* podem ser aplicadas na base de dados em lotes, aumentando a performance da aplicação, reduzindo o tráfego na rede, e optimizando a utilização do SGBD.

As classes que compõem o ADO.NET dividem-se em dois grupos: as genéricas, que são independentes do SGBD utilizado, e as restantes, que são específicas do SGBD a acederem.

No primeiro grupo temos, entre outras, as seguintes classes, pertencentes ao *namespace* **System.Data**:

- **DataSet** — representa um *buffer* local.
- **DataTable** — representa uma tabela com dados, organizados em linhas e colunas.
- **DataRow** — representa um registo (linha) de uma tabela (*DataTable*).
- **DataColumn** — representa um campo (coluna) de uma tabela.
- **DataRelation** — representa uma relação entre diferentes tabelas do *DataSet*.

No segundo grupo temos classes que representam ligações a SGBD's, comandos SQL, etc. A selecção das classes a utilizar depende do SGBD a que pretendemos aceder. Caso o SGBD seja o Microsoft Access, podemos utilizar as seguintes classes do *namespace* **System.Data.OleDb**:

- **OleDbConnection** — representa uma ligação ao SGBD.
- **OleDbCommand** — representa um comando SQL.
- **OleDbDataAdapter** — é uma classe que permite operar no modo desconectado; controla a ligação ao SGBD e a transferência/actualização de dados entre o *dataset* e a base de dados.
- **OleDbDataReader** — permite realizar *queries* em bases de dados (no modo tradicional) e ler os registos retornados.
- **OleDbParameter** — representa um parâmetro dum comando SQL ou de um *stored procedure*.

## Acesso a bases de dados no modo tradicional

Nesta secção vamos aceder a uma base de dados Microsoft Access no modo tradicional, isto é, sem recurso a um *dataset*, e controlando explicitamente a ligação ao SGBD.

Vamos começar por criar um método para estabelecer uma ligação ao SGBD e seleccionar uma base de dados contida no ficheiro indicado. Aproveitamos para criar também um método para fechar a ligação à base de dados.

```
OleDbConnection Ligar(string ficheiro)
{
    string connectionString =
        "provider=Microsoft.Jet.OLEDB.4.0;data source=" + ficheiro;
    OleDbConnection con = new OleDbConnection(connectionString);
    con.Open();
    return con;
}

void Desligar(OleDbConnection con)
{
    con.Close();
}
```

O método *Ligar* retorna um objecto do tipo *OleDbConnection*, que representa a ligação ao SGBD. A variável *connectionString* indica os parâmetros necessários para estabelecer a ligação. O formato da *connectionString* varia de SGBD para SGBD. O site <http://www.connectionstrings.com/> indica o formato específico da *connectionString* a utilizar em cada um dos principais SGBD's.

Uma vez estabelecida a ligação à base de dados com o método *Open*, podemos executar comandos SQL. O método apresentado a seguir utiliza objectos *OleDbCommand* e *OleDbDataReader* para ler um conjunto de registos da base de dados, que são depois escritos na consola. Os dados a ler provêm da tabela *Alunos* e são obtidos com um comando SQL *select*.

```

void ListarAlunos(OleDbConnection con)
{
    string cmdString = "select Num, Nome from Alunos";
    OleDbCommand cmd = new OleDbCommand(cmdString, con);
    OleDbDataReader dr = cmd.ExecuteReader();
    while (dr.Read())
    {
        Console.WriteLine("Aluno n.º {0} - {1}",
            dr.GetValue(0), dr.GetValue(1));
    }
    dr.Close();
}

```

São seleccionados e listados todos os registos da tabela *Alunos* (campos *Num* e *Nome*). O método *Read* do objecto *OleDbDataReader* retorna *true* enquanto houver registos para ler. O método *GetValue* permite obter o valor dos campos do registo actual. Neste exemplo, os campos são referidos pelo seu índice. Para referir os campos pelo seu nome, poderíamos utilizar o seguinte código:

```

Console.WriteLine("Aluno n.º {0} - {1}", dr["Num"], dr["Nome"]);

```

Adicionalmente, a classe *OleDbDataReader* contém métodos que retornam os valores dos campos representados segundo os tipos de dados básicos: *GetBoolean*, *GetByte*, *GetDouble*, *GetInt32*, etc.

De seguida é apresentado um exemplo de uma *query* mais complexa, que envolve duas tabelas:

```

void ListarTurmas(OleDbConnection con)
{
    string cmdString = "select T.Ano, T.Letra, P.Nome " +
        "from Turmas T, Professores P " +
        "where T.DirectorTurma = P.CodProfessor";
    OleDbCommand cmd = new OleDbCommand(cmdString, con);
    OleDbDataReader dr = cmd.ExecuteReader();
    while (dr.Read())
    {
        Console.WriteLine("Turma {0}.º {1}; DT: {2}",
            dr["Ano"], dr["Letra"], dr["Nome"]);
    }
    dr.Close();
}

```

As operações de inserção, remoção e actualização de dados são mais simples, já que não envolvem a leitura de dados.

O método apresentado a seguir permite remover um aluno identificado pelo seu número. O comando é executado com o método *ExecuteNonQuery*, que retorna o número de registos afectados pelo comando (registos inseridos, removidos ou alterados).

```
void RemoverAluno(OleDbConnection con, int num)
{
    string cmdString = "delete from Alunos where Num = @num";
    OleDbCommand cmd = new OleDbCommand(cmdString, con);
    cmd.Parameters.AddWithValue("@num", num);
    int n = cmd.ExecuteNonQuery();
    Console.WriteLine("Foram removido(s) {0} alunos", n);
}
```

O comando SQL utilizado recebe um parâmetro de nome *@num*, cujo valor é indicado com o método *AddWithValue* aplicado ao objecto retornado pela propriedade *Parameters*. A utilização de parâmetros para indicar valores literais em comandos SQL é preferível, por questões de segurança, em relação à alternativa que consiste em escrever directamente esses valores nos comandos, como no exemplo apresentado a seguir:

```
string cmdString = "delete from Alunos where Num = " + num.ToString();
```

A actualização de um ou mais registos de dados é feita de forma similar à remoção de registos. A diferença está no comando SQL utilizado.

O método apresentado a seguir permite mudar o nome a um aluno identificado por número.

```
void MudarNomeAluno(OleDbConnection con, int num, string novoNome)
{
    string cmdStr = "update Alunos set Nome = @nome where Num = @num";
    OleDbCommand cmd = new OleDbCommand(cmdStr, con);
    cmd.Parameters.AddWithValue("@nome", novoNome);
    cmd.Parameters.AddWithValue("@num", num);
    int n = cmd.ExecuteNonQuery();
    Console.WriteLine("Foram alterados {0} registos", n);
}
```

A operação de inserção de dados é realizada com o comando SQL *insert*, e funciona de forma similar às operações de remoção e actualização de dados.

A classe *OleDbCommand* inclui um terceiro método para executar comandos, para além dos métodos *ExecuteReader* e *ExecuteNonQuery*. O terceiro método chama-se *ExecuteScalar*, e deve ser utilizado para executar *queries* que retornam um único registo com um único campo (isto é, um valor escalar). Um exemplo de aplicação desse método é apresentado a seguir. O objecto da *query* utilizada é obter o número de alunos de determinada turma (identificada por ano e letra; por exemplo, 9º A).

```

void LerNumeroDeAlunos(OleDbConnection con, int ano, string letra)
{
    string cmdString = "select count(*) from Alunos " +
        "where Ano = @ano and Letra = @letra";
    OleDbCommand cmd = new OleDbCommand(cmdString, con);
    cmd.Parameters.AddWithValue("@ano", ano);
    cmd.Parameters.AddWithValue("@letra", letra);
    int n = (int) cmd.ExecuteScalar();
    Console.WriteLine("Existem {0} alunos na turma {1}ª {2}",
        n, ano, letra);
}

```

A abertura e fecho da ligação ao SGBD devem ser efectuados em função do SGBD e dos clientes que o utilizam, e do modo de funcionamento da aplicação. Caso se trate de um SGBD e de uma base de dados utilizados por um único cliente, não é necessário fechar a ligação ao SGBD nos intervalos entre a execução de comandos SQL. Mas caso se trate de um SGBD acedido por vários clientes, e que tenha que suportar um elevado número de acessos, é mais proveitoso manter a ligação ao SGBD aberta o mínimo de tempo possível.

### Acesso a bases de dados no modo desconectado

A utilização do ADO.NET no modo desconectado implica a utilização de um objecto que vai gerir um *buffer* local de dados (um *dataset*), e controlar a ligação ao SGBD. No caso de uma base de dados Microsoft Access utiliza-se um objecto *OleDbDataAdapter*.

Vamos começar por criar um método de inicialização do acesso à base de dados. O método cria o objecto que representa a ligação ao SGBD e depois cria um *dataset*.

```

void InicializarBD(string filename, out OleDbConnection con,
    out DataSet ds)
{
    string conString = "provider=Microsoft.Jet.OLEDB.4.0;data source=" +
        filename;
    con = new OleDbConnection(conString);
    ds = new DataSet();
}

```

De seguida criamos uma classe para lidar com a tabela *Alunos*. A classe é apresentada a seguir (por agora apenas com variáveis e um construtor).

```

class TabAlunos
{
    OleDbConnection con;

```

```

OleDbDataAdapter da;

DataSet ds;

DataTable dt;

public TabAlunos(OleDbConnection con, DataSet ds)
{
    const string cmdString = "select Num, Nome from Alunos";
    this.con = con;
    this.ds = ds;

    // criar o objecto OleDbDataAdapter e preencher o dataset
    // com os dados da tabela Alunos
    da = new OleDbDataAdapter(cmdString, con);
    da.Fill(ds, "alunos");
    dt = ds.Tables["alunos"];

    // indicar que a chave primária é o campo Num
    // (é necessário para utilizar o método Find)
    dt.PrimaryKey = new DataColumn[1] { dt.Columns["Num"] };

    // criar os comandos a utilizar para
    // inserir/remover/actualizar dados
    // os valores dos parâmetros dos comandos provêm dos
    // campos da tabela

    OleDbCommand cmd = new OleDbCommand(
        "update Alunos set Nome = @nome where Num = @num", con);
    cmd.Parameters.Add("@nome", DbType.Char, 0, "Nome");
    cmd.Parameters.Add("@num", DbType.Integer, 0, "Num");
    da.UpdateCommand = cmd;
}

```

```

cmd = new OleDbCommand(
    "insert into Alunos (Num, Nome) values (@num, @nome)", con);
cmd.Parameters.Add("@num", OleDbType.Integer, 0, "Num");
cmd.Parameters.Add("@nome", OleDbType.Char, 0, "Nome");
da.InsertCommand = cmd;

cmd = new OleDbCommand(
    "delete from Alunos where Num = @num", con);
cmd.Parameters.Add("@num", OleDbType.Integer, 0, "Num");
da.DeleteCommand = cmd;
}

// restantes métodos (apresentados mais à frente)
}

```

O construtor da classe *TabAlunos* executa a query que vai preencher o *dataset*, criando uma tabela local de nome *alunos*. O objecto *OleDbDataAdapter* vai gerir a ligação entre esta tabela local em memória e a tabela correspondente na base de dados. Para que o objecto possa gerir as actualizações, as remoções e as inserções de registos, é necessário indicar os respectivos comandos.

O método apresentado a seguir permite inserir um novo aluno, fazendo com que o registo criado para o novo aluno seja guardado na base de dados.

```

public void InserirAluno(int num, string nome)
{
    // criar um novo registo à tabela local alunos
    // são indicados os valores para os campos Num e Nome
    dt.Rows.Add(num, nome);
    // executar o comando SQL insert para inserir o registo
    da.Update(ds, "alunos");
    // guardar as alterações na base de dados (commit)
    ds.AcceptChanges();
}

```

O método apresentado a seguir permite remover um aluno com determinado número. Para localizar o registo é utilizado o método *Find*, que permite localizar um registo indicando-se os valores para os campos que compõem a chave primária. Neste caso, a chave primária é composta pelo campo *Num*. Se pretendessemos localizar um registo indicando uma condição mais complexa, possivelmente envolvendo outros campos, poderíamos utilizar o método *Select* do objecto *DataTable*.

```

public void RemoverAluno(int num)
{
    // localizar o registo com o método Find
    DataRow row = dt.Rows.Find(num);
    // remover o registo
    row.Delete();
    // executar o comando SQL delete e guardar as alterações
    da.Update(ds, "alunos");
    ds.AcceptChanges();
}

```

O método apresentado a seguir exemplifica a utilização do método *Select* do objecto *DataTable*, e serve para encontrar um valor adequado para o número de um novo aluno (não pode ser igual a um já existente). É procurado o valor máximo no campo *Num* e é retornado esse valor incrementado de uma unidade. Se a pesquisa não retornar um registo é porque ainda não existem alunos; nesse caso, o novo aluno vai assumir o número 1.

```

public int ProximoNumero()
{
    int proxNum = 1;
    // procura o registo com o valor máximo no campo Num
    DataRow[] rows = dt.Select("Num = max(Num)");
    // se retornou um registo usa o valor do campo Num do
    // registo retornado + 1
    // caso contrário, não existem alunos, e é utilizado o valor 1
    if (rows.Length == 1) proxNum = 1 + (int)rows[0]["Num"];
    return proxNum;
}

```

O método seguinte permite alterar o nome de um aluno identificado pelo seu número. São utilizados os métodos *BeginEdit* e *EndEdit* do objecto *DataRow* (que representa o registo) para delimitar as alterações ao registo.



```

public void ActualizarAluno(int num, string novoNome)
{
    // localizar o registo do aluno
    DataRow row = dt.Rows.Find(num);
    // iniciar a edição
    row.BeginEdit();
    // alterar o nome do aluno
    row["Nome"] = novoNome;
    // finalizar a edição
    row.EndEdit();
    // executar o comando SQL update e guardar as alterações na BD
    da.Update(ds, "alunos");
    ds.AcceptChanges();
}

```

A classe *TabAlunos* fica completa com as propriedades apresentadas a seguir. A primeira propriedade retorna o número de registos na tabela; a segunda retorna o objecto *DataTable* que representa a tabela, para que possa ser utilizado fora dos métodos da classe.

```

public int NumRegistos
{
    get { return dt.Rows.Count; }
}

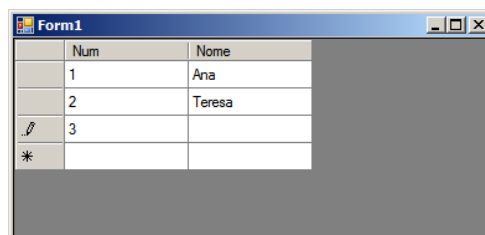
public DataTable Tabela
{
    get { return dt; }
}

```

#### Controle DataGridView

O controle *DataGridView* pode ser utilizado para manipular dados de vários tipos de origens, incluindo *arrays* e objectos *DataTable*.

A *form* apresentada a seguir inclui um controle *DataGridView* ligado à tabela *Alunos*.



O código da *form* necessário para ligar o controle à tabela e fazer com que tudo funcione bem, incluindo as actualizações na base de dados, é apresentado a seguir.

```

// declaração de variáveis

OleDbConnection con;

OleDbDataAdapter da;

DataSet ds;

DataTable dt;

// fazer as inicializações quando a form é carregada

private void Form1_Load(object sender, EventArgs e)
{
    AbrirBD(@"c:\bd1.mdb");
}

// atualizar a base de dados quando a form é fechada
// (e a aplicação é encerrada)
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    ActualizarBD();
}

// fazer as inicializações necessárias, incluindo criar os
// comandos de update, insert e delete
void AbrirBD(string filename)
{
    string conString = "provider=Microsoft.Jet.OLEDB.4.0;data source=" +
        filename;
    con = new OleDbConnection(conString);

    const string cmdString = "select Num, Nome from Alunos";
    da = new OleDbDataAdapter(cmdString, con);

    ds = new DataSet();
    da.Fill(ds, "alunos");
    dt = ds.Tables["alunos"];

    OleDbCommand cmd = new OleDbCommand(
        "update Alunos set Nome = @nome where Num = @num", con);

```

```

cmd.Parameters.Add("@nome", OleDbType.Char, 0, "Nome");
cmd.Parameters.Add("@num", OleDbType.Integer, 0, "Num");
da.UpdateCommand = cmd;

cmd = new OleDbCommand(
    "insert into Alunos (Num, Nome) values (@num, @nome)", con);
cmd.Parameters.Add("@num", OleDbType.Integer, 0, "Num");
cmd.Parameters.Add("@nome", OleDbType.Char, 0, "Nome");
da.InsertCommand = cmd;

cmd = new OleDbCommand("delete from Alunos where Num = @num", con);
cmd.Parameters.Add("@num", OleDbType.Integer, 0, "Num");
da.DeleteCommand = cmd;

// ligar a tabela ao controle
dataGridView1.AutoGenerateColumns = true;
dataGridView1.DataSource = dt;
}

// atualizar a base de dados
void AtualizarBD()
{
    da.Update(ds, "alunos");
    ds.AcceptChanges();
}

```

## Referências

- <http://www.softsteel.co.uk/tutorials/cSharp/>
- <http://www.csharp-station.com/Tutorial.aspx>
- <http://csharpcomputing.com/Tutorials/TOC.htm>
- [http://msdn2.microsoft.com/en-us/library/9b9dty7d\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/9b9dty7d(VS.80).aspx)
- [http://msdn2.microsoft.com/en-us/library/2s05feca\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/2s05feca(VS.80).aspx)
- C# School – Programmers Heaven (cf. Site da Disciplina)
- Microsoft ADO.Net - Step by Step. Rebecca M. Riordan. Microsoft Press, 2002.
- Programming Microsoft ADO.NET 2.0 Applications: Advanced Topics. Glenn Johnson. Microsoft Press, 2006.

## Parte XII – Persistência

Autor: Luís Ferreira

---

Esta parte do documento continua a abordagem da *Serialização e Persistência* em C#.

### Sumário:

#### Considerações

Serializar deve ser visto como a forma de preservar a informação das diferentes instâncias de objectos de uma aplicação, ie, garantir que, entre duas sessões de trabalho, a informação permanece. Em termos práticos é a forma de guardar a informação (objectos), normalmente em ficheiro<sup>13</sup>.

Em .NET Framework existem três formas de serialização: em ficheiro binário, em ficheiro XML, em SOAP (Simple Object Access Protocol) ou mesmo para Bases de Dados (relacional ou não).

No capítulo desta sebeta que aborda o “Tratamento de Ficheiros” muitos dos conceitos essenciais já foram apresentados. Mas iniciar um processo de serialização, é fundamental entender a forma como um objecto pode ser armazenado numa estrutura diferente, dita, Persistente.

#### Mapeamento de Objectos para Bases de Dados

Mapear refere-se à forma de guardar os objectos e respectivas relações numa base de dados relacional (ou não). Basicamente, significa “passar” de um modelo de classes para um modelo físico de armazenamento de dados. No nosso caso vamos abordar Bases de Dados Relacionais

---

<sup>13</sup> Pode também ser efectuada para Base de Dados

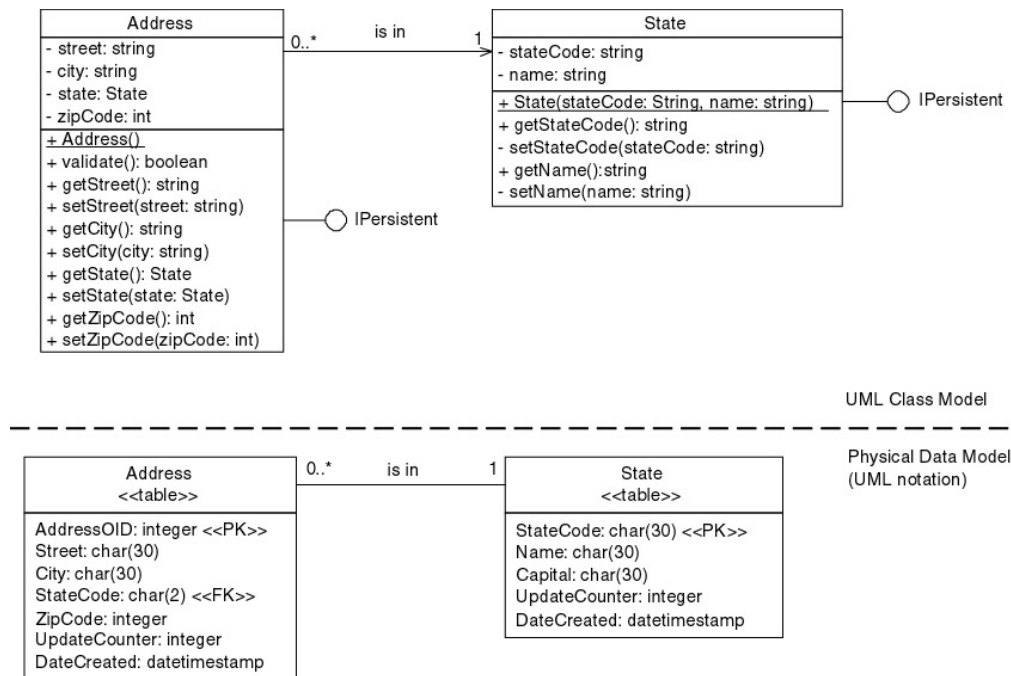


Figura 29 – De Modelo Classe para Modelo Físico

Na Figura 1 mostra-se, numa notação UML, um exemplo de mapeamento de um modelo lógico de classes para um modelo físico correspondente, de uma base de dados relacional.

**Regra 1:** Um atributo de uma classe é mapeado para zero ou mais colunas de uma tabela

No nosso caso temos, por exemplo, o mapeamento do atributo *Address.street* para a coluna *Street* da tabela *Address*.

Pode acontecer que os atributos de um objecto sejam eles próprios objectos. No nosso caso, *Address.state* é um objecto *State*. Implicitamente existe uma relação entre estas duas classes.

Também convém relembrar que nem toda a informação poderá interessar guardar (ou carregar) na(da) base de dados (assunto visto aquando a *Serialização*). Por exemplo, um atributo *mediaFinal* de um objecto *Aluno*, poderá não interessar guardar pois é um valor calculado a partir de outros. No nosso exemplo, a coluna *Capital* da tabela *State* não tem correspondente na classe *State*.

Em situações extremas um atributo poderá ser mapeado em várias colunas de várias tabelas ou vice-versa.

## Shadow Information

Por vezes é necessário manter informação extra (shadow) no sentido de facilitar a gestão global de toda a informação. São os casos das chaves primárias das tabelas, contadores, controladores, temporizadores, etc. No nosso caso a coluna *AddressOID* (chave primária), *UpdateCounter* (flag de controlo) e *DateCreated* (registra a data da última alteração) não têm representantes na classe *Address*. Contudo poderá ser útil o seu mapeamento. A Figura 2 mostra uma versão actualizada das nossas classes *Address* e *State* (onde se representam só os atributos).

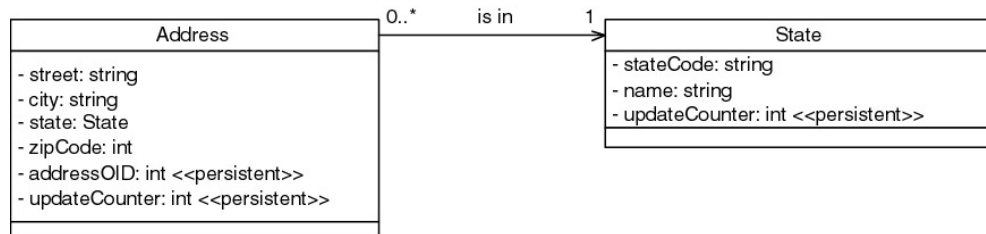


Figura 30 - Mapeamento de informação Extra

**Regra 2:** atributos shadow são mapeados com acesso *private* e *persistentes*

Foram então adicionados os atributos *Address.addressOID*, *Address.updateCounter* e *State.updateCounter*. O atributo *State.stateCode* já estava presente. Os atributos *DateCreated* não foram mapeados pois são valores que não serão “alterados” durante a manipulação dos objectos em memória. Mas caso essa informação fosse necessária durante a aplicação, tal deveria ser também mapeado.

**Regra 3:** atributos shadow são só de leitura

Os valores dos atributos shadow deverão manipulados aquando o “carregamento” de dados da BD e nunca mais alterados pela aplicação.

## Mapeamento de Herança de classes

Existem três possibilidades de preservar a herança de classes:

- Mapear toda a hierarquia numa única tabela
- Mapear cada classe concreta numa tabela própria
- Mapear cada classe numa tabela própria

A Figura 3 mostra uma hierarquia de classes simples (Pessoa → Estudante : Professor) e as três formas apresentadas de preservar essa informação.

Algumas observações:

- As chaves das tabelas - *PersonOID*, *StudentOID*, *ProfessorOID* - correspondem todas ao atributo *oid* da classe *Person*;
- Na tabela única (*Person*) foi acrescentado a coluna *TypeCode* para distinguir que tipo de pessoa está registado (student, professor, etc...) em cada linha da tabela;
- Na estratégia “Uma Tabela por Classe”, as tabelas *Student* e *Professor* possuem como chave primária *PersonID*, que ao mesmo tempo é uma chave estrangeira (FK) para a tabela *Person*;

Na generalidade dos casos a estratégia de uma única tabela é a mais utilizada. A tabela seguinte resume as principais considerações sobre as três propostas:

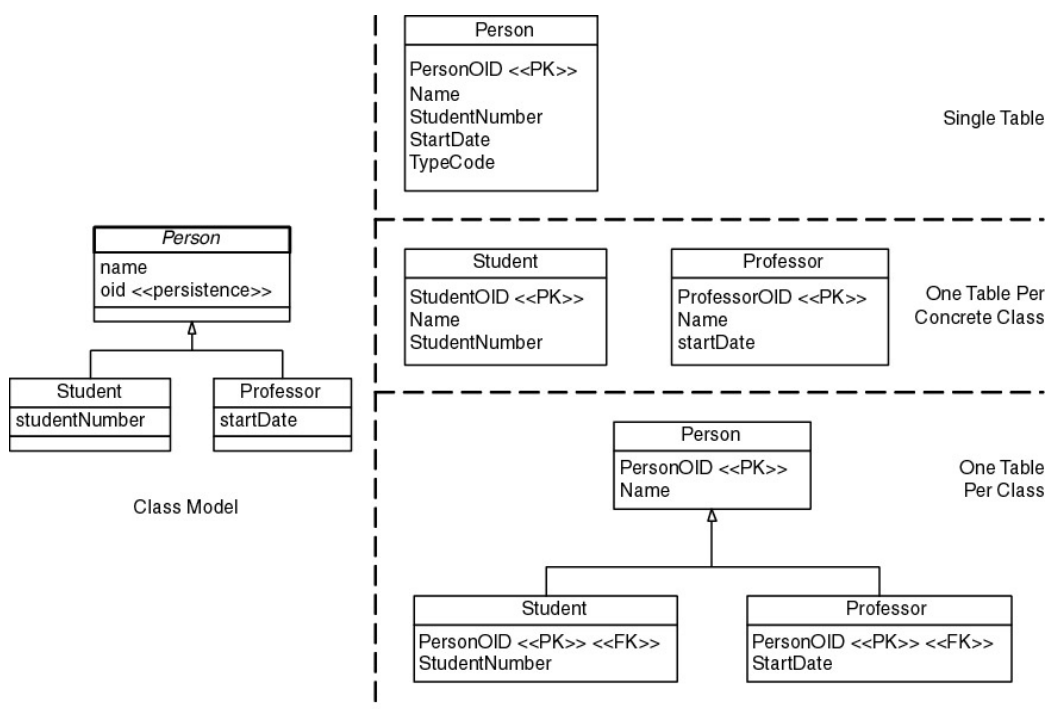


Figura 31 - Preservar Hierarquias de classes

Estratégia	Vantagens	Desvantagens
<b><i>Tabela Única</i></b>	Mais simples	Alguma complexidade devido ao agrupamento numa única tabela de toda a hierarquia. Qualquer alteração na hierarquia de classes poderá implicar importantes alterações na tabela!
	Fácil de acrescentar mais classes. Corresponderá a acrescentar novas colunas.	Poderá haver espaço perdido na BD. Algumas colunas não terão conteúdo!
	Suporta Polimorfismo. Basta alterar o tipo da coluna na tabela!	Pode causar confusão na utilização de flags. (ex. pessoa pode ser professor e aluno)
	Acesso mais rápido aos dados. Uma só tabela!	Tabela pode crescer exponencialmente!

<b><i>Uma tabela por cada classe concreta</i></b>	Rápido o acesso à informação de um determinado objecto.	Modificações nas classes podem implicar alterar todas as tabelas envolvidas. Por exemplo, ao acrescentar <i>birthDate</i> à classe <i>Person</i> , tem de criar a respectiva coluna nas tabelas <i>Person</i> , <i>Student</i> e <i>Professor</i> .
---	---	---

<b><i>Uma tabela por Classe</i></b>	Fácil mapear pois existe uma relação 1-para-1	Muitas tabelas na BD
	Suporta Polimorfismo de objectos, uma vez que cada objecto de cada tipo está na sua respectiva tabela.	Demora na leitura e escrita de dados pois tem de aceder a múltiplas tabelas
	Fácil de modificar a hierarquia pois corresponde a adicionar ou remover tabelas	



	A quantidade de dados aumenta com o número de objectos;	
--	---	--

## Mapear Relações

Ao referirmo-nos a Relações, englobamos Associações e Composições de classes. Em classes as relações são conseguidas através de referência a objectos e métodos, enquanto que em BD são implementadas através de chaves estrangeiras (FK – foreign keys).

O mapeamento de relações depende da sua cardinalidade. Existem essencialmente três tipos: *Um-para-Um*, *Um-para-N* e *N-para-M*. Vejamos cada um dos casos.

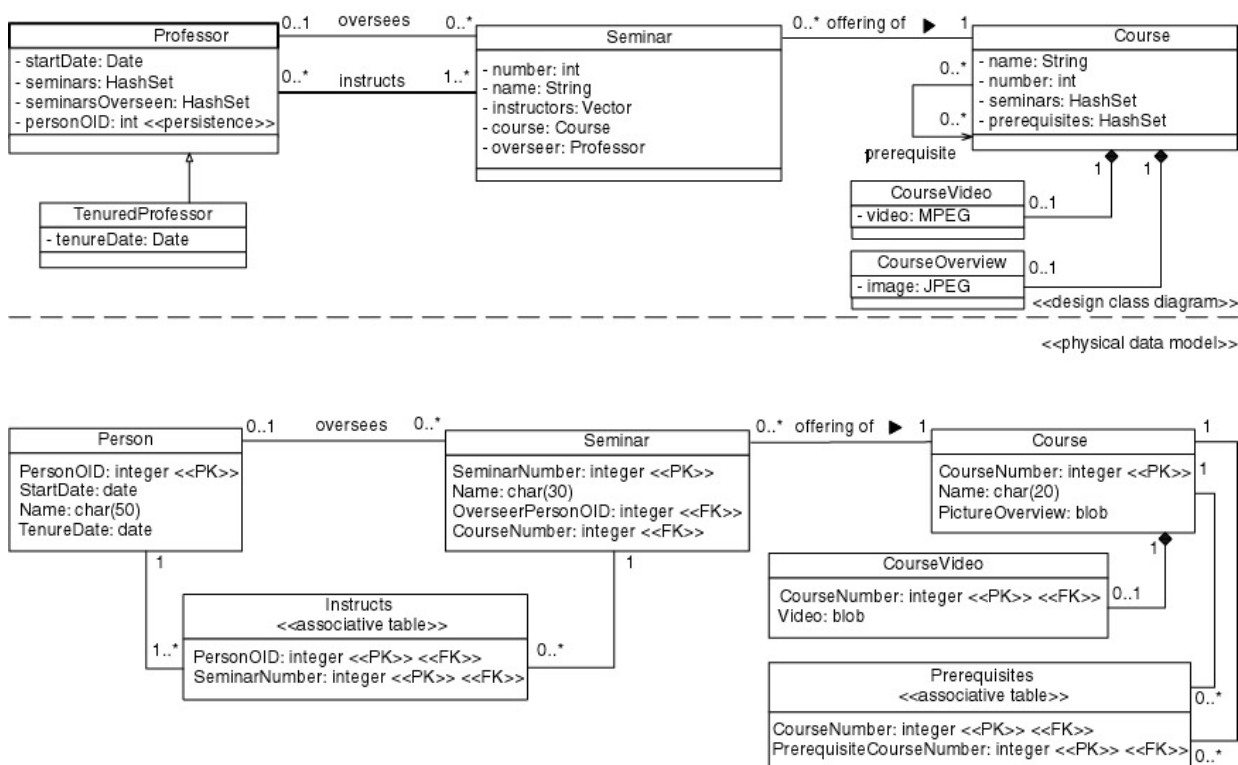


Figura 32- Mapeamento de Relações

**Um-para-Um:** Podem ser aplicadas duas estratégias. Se cada classe corresponde a uma tabela, como no caso das classes *Course* e *CourseVideo* da Figura 4, então deve existir uma FK numa das tabelas. Neste caso adicionou-se a FK *CourseNumber* na tabela *CourseVideo* (embora se pudesse colocar em qualquer uma das tabelas). A segunda possibilidade pode ser através do mapeamento de ambas as classes numa só tabela, tal foi feito nas classes *Course* e *CourseOverview*, que foram ambas mapeadas na tabela *Course* (coluna *PictureOverview*).

**Um-para-N:** As relações do tipo *Um-para-N* (leia-se Um para Muitos) são implementadas em classes através de Collections, tais como HashSets no lado “Muitos” da relação e uma referência a Objecto no lado do “Um”. No nosso caso, *Course.seminars* e *Seminar.course*, respectivamente, na relação entre *Course* e *Seminar*.

**N-para-M:** As relações do tipo *N-para-M* (leia-se Muitos para Muitos) entre duas classes são mapeadas para uma tabela associativa. Este tipo de tabela (resultante geralmente de um processo de Normalização) comporta as chaves primárias de ambas as tabelas relacionadas. No nosso exemplo, a relação *instructs* entre as classes *Professor* e *Seminar*, é mapeada na tabela *Instructs* que relaciona as tabelas *Seminar* com *Person*. Na prática criaram-se duas relações *Um-para-N*.

## Implementar Persistência

Existem essencialmente três formas de implementar a persistência de objectos:

**Directamente:** Os objectos acedem directamente aos dados através de queries SQL à Base de Dados. Isto consegue-se através de drivers específicos das BD utilizadas ou mesmo drivers genéricos, tipo JDBC, ADO.NET ou LINQ. Desta forma é fácil e rápido implementar operações CRUD sobre a BD. No entanto o código produzido está demasiado “preso” ao esquema da BD. Pequenas alterações na BD poderá implicar alterações significativas no código produzido. Útil para Protótipos ou pequenas aplicações.

**Objectos de Acesso a Dados (DAO):** São criadas classes específicas para interagir pelos objectos com a Base de Dados, ie, são responsáveis por encapsular todas as operações que o objecto necessita da BD. A este conjunto de classes dá-se geralmente o nome de *DAL – Data Access Layer*. Por exemplo, uma classe *Student* deverá ter uma classe *StudentData* que garante todas as operações SQL com a BD. Indicado para protótipos, soluções pequenas (<=50 classes) ou BD legadas. Mais independentes do modelo de dados. Alterações na BD poderão no entanto implicar a reescrita destas classes. As classes DAO poderão ser codificadas manualmente

**Ferramentas específicas:** Apelidadas de *Persistence Frameworks*, conseguem encapsular por completo todas as operações de manipulação da BD por parte dos objectos. Em vez de se codificar, define-se um conjunto de metadados que definem as relações e mapeamentos. Uma lista de ferramentas deste tipo pode ser encontrada em <http://www.ambysoft.com/persistenceLayer.html>. *NHibernate* (<http://www.hibernate.org/>) é uma das ferramentas mais utilizadas em .NET. Aqui, o utilizador pode abstrair-se por completo da forma como os seus objectos são “armazenados”. A sua performance é superior à da escrita directa de queries SQL.

## Bases de Dados Orientadas a Objectos

A escolha de uma Base de Dados Orientada a Objectos (ODBMS), significa que não é necessário ter estes cuidados de mapeamento de objectos. Num ODBMS nativo, o objecto persistente é igual aquele que fora criado na linguagem de programação orientada a objectos, tipo C#. O programador pode focar-se somente na vertente

lógica de todo o sistema. Uma das bases de dados orientadas a objectos nativas e *open-source* mais populares é o *db4o* (<http://www.db4o.com/>).

### Boa conduta em Programação

- Escrever sempre código limpo (XP- Extreme Programming)
- *Refactoring* deve ser fácil e simples
- Pensar antes de codificar
- Desenvolver em pequenos passos
- Escrever código perceptível
- Seguir standards ou design patterns
- Documentar para facilitar a percepção do código
- Documentar antes de codificar
- Seguir a regra dos 30 segundos (no máximo em 30 s consegue-se perceber o que um módulo faz)
- Implementar métodos pequenos
- Optimizar código como última fase do desenvolvimento

### Exemplo

Vamos de seguida analisar um excerto de código de implementar persistência para uma base de dados orientada a objectos. A base de dados utilizada foi o *db4o*.

É necessário fazer referência a `Namespace Db4objects.Db4o` e `Db4objects.Db4o.Query`. Estes módulos correspondem a DLLs que ficam disponíveis após a instalação do *db4o*.

Neste exemplo é criada uma camada DAL muito simples no sentido de demonstrar a sua importância numa aplicação normal. Esta camada foi implementada na classe DAO.

Pretende-se, em primeiro lugar, gerir a informação referente a uma Pessoa e depois de um Curso, onde leccionam um conjunto de Docentes. O *db4o* disponibiliza três formas de interagir com a Base de Dados: via *Query By Example (QBE)*, via *Native Code* e via *API SODA*. Neste nosso exemplo demonstra-se a aplicação de QBE e SODA.

O código utilizado neste exemplo foi simplificado no sentido de tornar a aplicação perceptível, e segue os principais passos indicados no tutorial do *db4o*, que se recomenda analisar. Analisemos então o código.

O Diagrama das Classes definidas está apresentado na Figura 5.

O código completo deste exemplo encontra-se no WOC.

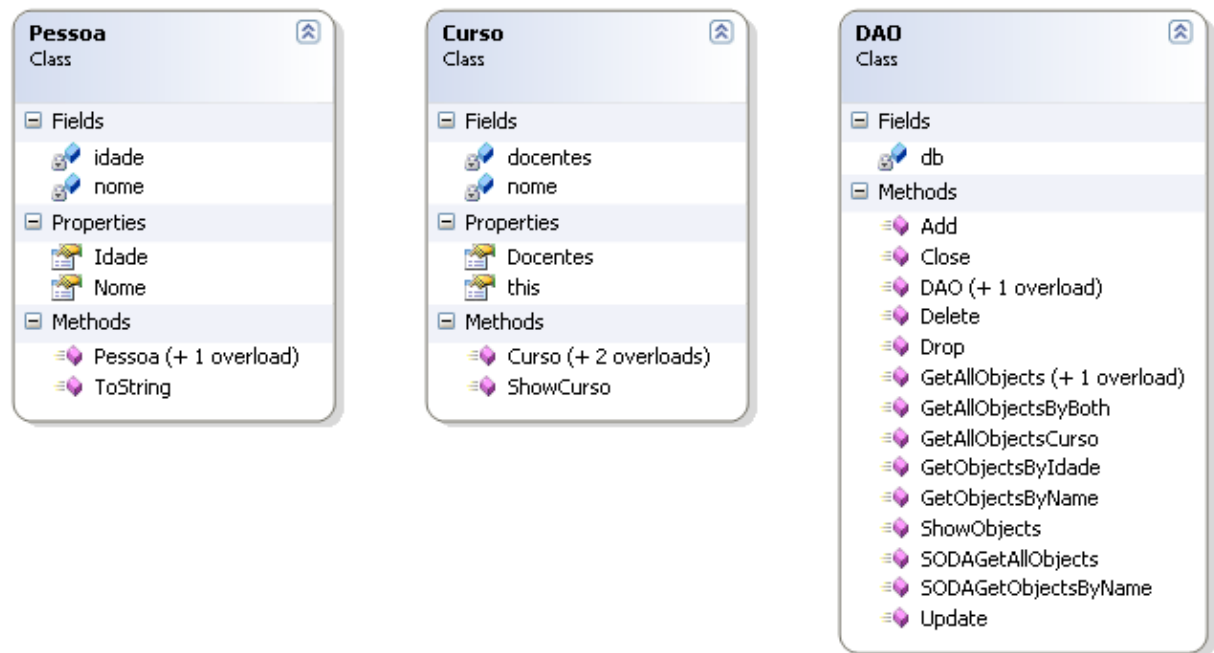


Figura 33- Diagrama de Classes do Exemplo

### Classe DAO

```
//by lufer
//POO - 2008
using System;
using System.Collections;
using System.Text;
using System.Windows.Forms;

using Db4objects.Db4o;
using Db4objects.Db4o.Query;
```

```
namespace OD
```

```
{
```

```
    /// <summary>
```

```
    /// db4o Data Layer Manipulation
```

```
    /// Implementaram-se as principais funções de interacção (CRUD) com a
```

```
    /// Base de Dados:
```

```

/// Create
/// Remove
/// Update
/// Delete
/// </summary>

```

```
public class DAO
```

```
{
```

```
    //Handler da Base de Dados
```

```
    IObjectContainer db;
```

```
#region Constructors
```

```
public DAO()
```

```
{
```

```
    db = Db4oFactory.OpenFile("default.db4o");
```

```
}
```

```
public DAO(string fileName)
```

```
{
```

```
    db = Db4oFactory.OpenFile(fileName);
```

```
}
```

```
#endregion
```

```
//-----
```

```
#region Methods_via_QBE
```

```
/// <summary>
```

```
/// Insere um objecto na BD
```

```
/// </summary>
```

```
/// <param name="o">Objecto a inserir</param>
```

```
public void Add(Object o){
```

```
    db.Set(o);
```

```
}
```

```
/// <summary>
```

```
/// Update é o mesmo que Add após ter sido alterado
```

```

/// </summary>
/// <param name="o">Objecto alterado a actualizar</param>
public void Update(Object o)
{
    db.Set(o);
}

```

```

/// <summary>
/// Eliminar um objecto
/// </summary>
/// <param name="o">Object</param>
public void Delete(Object o)
{
    db.Delete(o);
}

```

```

/// <summary>
/// Carrega todos os objectos da BD
/// </summary>
/// <returns>IObjectSet</returns>
public IObjectSet GetAllObjects()
{
    //new Pessoa(null,0) é um template para filtro
    //return db.Get(new Pessoa(null,0));
    //ou
    return db.Get(typeof(Pessoa));
}

/// <summary>
/// Carrega todos os objectos da BD
/// </summary>
/// <returns>IObjectSet</returns>
public IObjectSet GetAllObjects(Object x)
{
    //new Pessoa(null,0) é um template para filtro
    //return db.Get(new Pessoa(null,0));
    //ou

```

```

        return db.Get(x);
    }

    /// <summary>
    /// Carrega todos os objectos da BD
    /// </summary>
    /// <returns>IObjectSet</returns>
    public IObjectSet GetAllObjectsCurso()
    {
        return db.Get(typeof(Curso));
    }

    /// <summary>
    /// Carrega todos os objectos de um determinado nome e idade
    /// </summary>
    /// <param name="name">Nome</param>
    /// <param name="idade">Idade</param>
    /// <returns>Conjunto de Objectos</returns>
    public IObjectSet GetAllObjectsByBoth(string name, int idade)
    {
        return db.Get(new Pessoa(name, idade));
    }

    /// <summary>
    /// Carrega todos os objectos da BD com um determinado nome
    /// </summary>
    /// <param name="name">Nome</param>
    /// <returns>Conjunto de objectos</returns>
    public IObjectSet GetObjectsByName(string name)
    {
        return db.Get(new Pessoa(name, 0));
    }

    /// <summary>
    /// Carrega todos os objectos da BD com uma determinada idade
    /// </summary>
    /// <param name="idade">Idade</param>
    /// <returns>Conjunto de objectos</returns>

```

```

public IObjectSet GetObjectsByIdade(int idade)
{
    return db.Get(new Pessoa(null, idade));
}

```

```

/// <summary>
/// Mostra todos os objectos
/// </summary>
/// <param name="oSet"></param>
public void ShowObjects(IObjectSet oSet)
{
    foreach (object o in oSet)
    {
        MessageBox.Show(o.ToString());
    }
}

```

#endregion

#region Methods\_via\_SODA\_Query\_API

```

/// <summary>
/// Get All via SODA
/// </summary>
/// <returns></returns>
public IObjectSet SODAGetAllObjects() {
    IQuery q = db.Query();
    //todos Pessoa
    q.Constrain(typeof(Pessoa));
    return (q.Execute());
}

```

```

/// <summary>
/// Get All via SODA por nome
/// </summary>
/// <param name="nome"></param>
/// <returns></returns>
public IObjectSet SODAGetObjectsByName(string nome)
{
    IQuery q = db.Query();
}

```



```

q.Constrain(typeof(Pessoa));
q.Descend("nome").Constrain(nome);
//q.Descend("name").OrderDescending();

//Variantes:
//NOT
//q.Descend("name").Constrain(nome).Not();
//AND
//IConstraint ic = q.Descend("name").Constrain("pedro");
//q.Descend("idade").Constrain(12).And(ic);
//OR
//q.Descend("idade").Constrain(12).Or(ic);
//Comparação:
//q.Descend("idade").Constrain(29).Greater();
//Outros
//q.Descend("name").Constrain(nome).StartsWith(true);
//SORT
//q.Descend("name").OrderDescending();
return (q.Execute());
}

```

#endregion

```

/// <summary>
/// Fecha a BD
/// </summary>
public void Close()
{
    db.Close();
}

```

```

/// <summary>
/// Elimina a BD
/// </summary>
public void Drop()
{
    foreach (object o in db.Get(typeof(Pessoa)))
    {
        db.Delete(o);
    }
}

```

```

    }
}
}

```

### Classe Pessoa

```

//by lufer
//POO-2008
using System;
using System.Text;

```

```

namespace OD
{
    /// <summary>
    /// Classe simplificada
    /// </summary>
    class Pessoa
    {
        string nome;
        int idade;

        #region construtores

```

```

        public Pessoa()
        {
            nome = "";
            idade = 0;
        }

        public Pessoa(string nome, int idade)
        {
            this.nome = nome;
            this.idade = idade;
        }

```

```

        #endregion

```

```

        #region Propriedades

```

```

public string Nome
{
    get { return nome;}
    set {nome=value;}
}

public int Idade
{
    get { return idade; }
    set { idade = value; }
}

```

```
#endregion
```

```

public override string ToString()
{
    return "Nome= " + nome + " Idade= " + idade.ToString();
}
}
}

```

### Classe Curso

```

//by lufer
//POO-2008
using System;
using System.Collections;
using System.Text;

```

```

namespace OD
{
    /// <summary>
    /// Classe simplificada
    /// </summary>
    class Curso
    {
        string nome;
    }
}

```

```
ArrayList docentes=new ArrayList();
```

```
/// <summary>
/// Construtores
/// </summary>
public Curso()
{
    nome = "";
}

public Curso(string nome)
{
    this.nome = nome;
}

public Curso(string nome, ArrayList ar)
{
    this.nome = nome;
    this.docentes = ar;
}
```

```
public ArrayList Docentes
{
    get { return docentes; }
}
```

```
public void ShowCurso()
{
    Console.WriteLine("Curso=" + nome);
    foreach (Pessoa p in docentes)
    {
        Console.WriteLine(p.ToString());
    }
}
```

```
/// <summary>
/// Indexador
```

```

    /// </summary>
    /// <param name="x"></param>
    /// <returns></returns>
    public Pessoa this[int x]
    {
        get { return (Pessoa)docentes[x]; }
        set { docentes.Add(value); }
    }
}
}

```

**Classe do Interface da aplicação:**

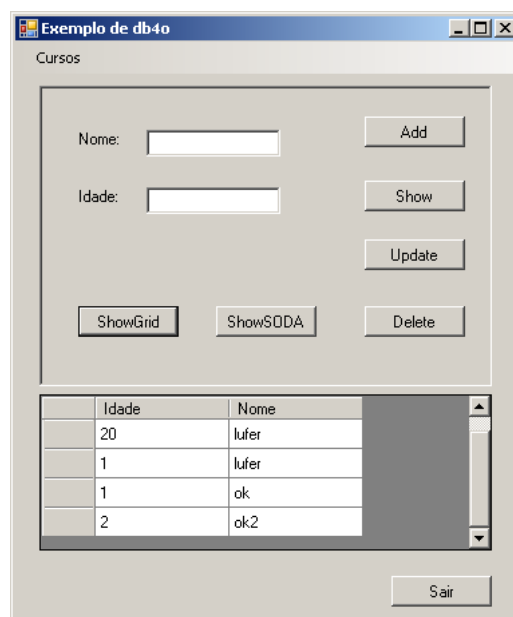


Figura 34 - Interface da aplicação exemplo

```

//by lufer
//POO-2008
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.IO;

```

```

using Db4objects.Db4o;

namespace OD
{
    public partial class FormPessoas : Form
    {
        DAO dao;

        public FormPessoas()
        {
            InitializeComponent();
            //File.Delete("TesteDB4O.bin");
            dao = new DAO("TesteDB4O.bin");
        }
    }
}

```

```

/// <summary>
/// Adicionar objecto
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button1_Click(object sender, EventArgs e)
{
    Pessoa p= new Pessoa(textBox1.Text,int.Parse(textBox2.Text));
    dao.Add(p);
    textBox1.Text = "";
    textBox2.Text = "";
}

```

```

/// <summary>
/// Sair
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button3_Click(object sender, EventArgs e)
{
    dao.Close();
}

```

```
this.Close();
}
```

```
/// <summary>
/// Filtrar
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button2_Click_1(object sender, EventArgs e)
{
    if (textBox1.Text == "" && textBox2.Text == "")
    {
        dao.ShowObjects(dao.GetAllObjects());
    }
    else
    {
        if (textBox1.Text == "")
        {
            dao.ShowObjects(dao.GetObjectsByIdade(int.Parse(textBox2.Text)));
        }
        else
        {
            if (textBox2.Text != "") //filtrar por nome e idade
            {
                dao.ShowObjects(dao.GetAllObjectsByBoth(textBox1.Text, int.Parse(textBox2.Text)));
            }
            else //filtrar por nome
            {
                dao.ShowObjects(dao.GetObjectsByName(textBox1.Text));
            }
        }
    }
}
```

```
/// <summary>
/// Update
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button4_Click(object sender, EventArgs e)
{
    IObjectSet os = dao.GetObjectsByName("lufar");

    //encontra o 1º
```

```

        Pessoa p = (Pessoa)os.Next();

        p.Idade *= 2;

        dao.Update(p);

    }

```

```

/// <summary>
/// Delete
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button5_Click(object sender, EventArgs e)
{
    IObjectSet os = dao.GetObjectsByIdade(1);
    Pessoa p = (Pessoa)os.Next();
    dao.Delete(p);
}

```

```

/// <summary>
/// Show
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button6_Click(object sender, EventArgs e)
{
    IObjectSet os;

    if (textBox1.Text == "" && textBox2.Text == "")
    {
        //os=dao.GetAllObjects();
        os = dao.GetAllObjects(new Pessoa(null,0));
    }
    else
    {
        if (textBox1.Text == "")
        {
            os=dao.GetObjectsByIdade(int.Parse(textBox2.Text));

```



```

    }
    else
    {
        if (textBox2.Text != "") //filtrar por nome e idade
            os=dao.GetAllObjectsByBoth(textBox1.Text, int.Parse(textBox2.Text));
        else //filtrar por nome
            os=dao.GetObjectsByName(textBox1.Text);
    }
    dataGridView1.DataSource = os;
}

```

```

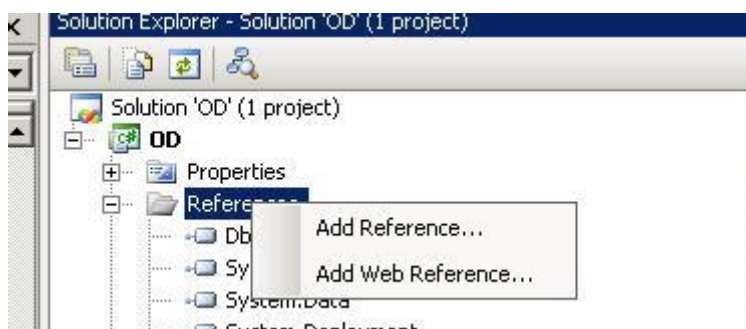
/// <summary>
/// SODA ShowGrid
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
private void button7_Click(object sender, EventArgs e)
{
    IObjectSet os = dao.SODAGetObjectsByName(textBox1.Text);
    dataGridView1.DataSource = os;
}

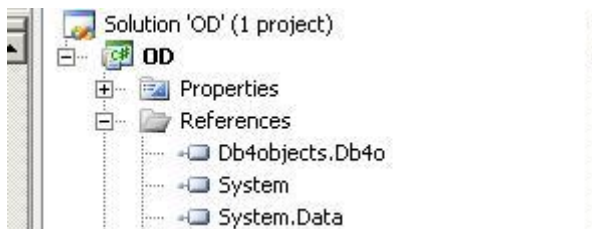
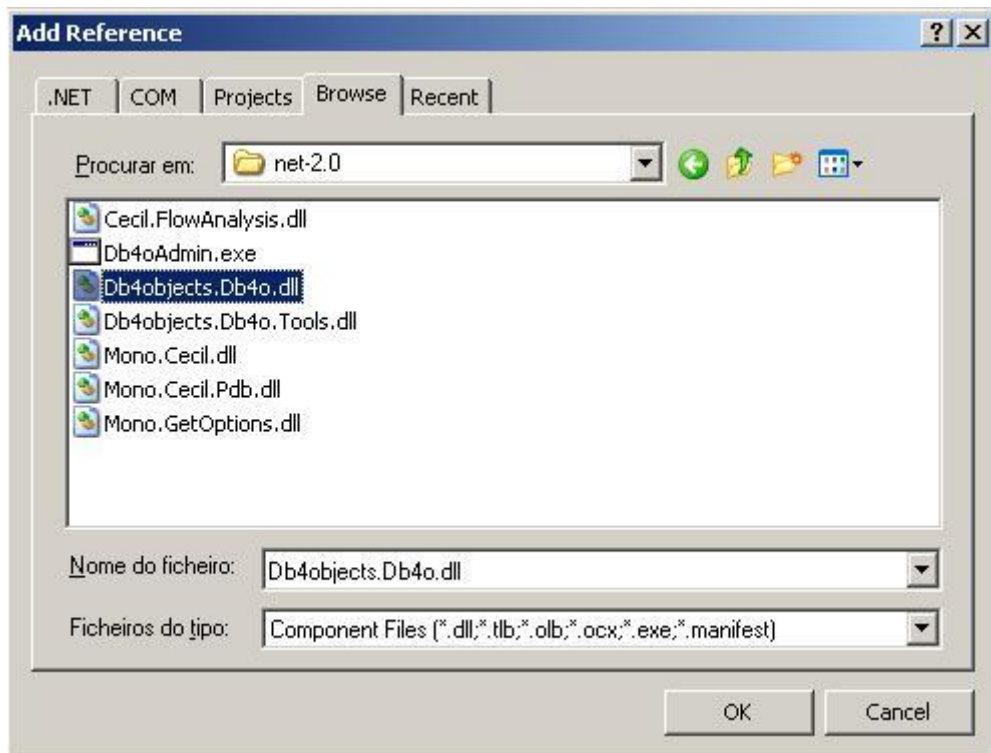
```

```

private void cursosToolStripMenuItem_Click(object sender, EventArgs e)
{
    new Cursos(dao).Show();
    //ou
    //new Cursos().Show();
}
}
}

```





## Referências

- The Object Primer, Third Edition, *Scott W. Ambler*
- Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, (Addison-Wesley)
- *Implement a Data Access Layer for Your App with ADO.NET* - <http://msdn.microsoft.com/en-us/magazine/cc188750.aspx>
- *Writing a Generic Data Access Component* - <http://www.c-sharpcorner.com/UploadFile/maresh/GenericDataProvide11192005012957AM/GenericDataProvide.aspx>
- .NET Developer's Journal - <http://dndj.sys-con.com/>
- [http://en.wikipedia.org/wiki/Object\\_database](http://en.wikipedia.org/wiki/Object_database)

## Parte XIII – Delegates e Eventos

Autor: Luís Ferreira

---

Esta parte do documento continua a abordagem de Delegates e Eventos em C#.

(em desenvolvimento)

### **Sumário:**

### **Considerações**

(Pro C# 2008 and the .NET 3.5 Platform)

## Parte XIV – Ofuscação e SVN

Autor: Luís Ferreira

---

Esta parte do documento aborda a Gestão de versões de código (CVS/SVN) e técnicas de ofuscação de código  
(em desenvolvimento)

## Índex

Ficheiros

OpenFileDialog, **146**

SaveFileDialog. *Consulte*