

# Projecto de Sistemas Industriais

---

**C#.NET**



- Variáveis
- Documentação XML
- *C# Preprocessor Directives*
- Operadores
- Estruturas de Decisão
- Estruturas de Ciclo
- Erros e Excepções
- Classes e Objectos

# C# - Orientada a Objectos

- Apresentada como primeira linguagem “Orientada por componentes” da família C/C++
- Considerações de evolução de versões pensadas na linguagem
- Componentes auto-descritivos:
  - Metadados, incluindo atributos definidos pelo utilizador, consultados em tempo de execução através de reflexão;
  - Documentação integrada em XML;
- Suporte para propriedades, métodos e eventos
- Programação simples:
  - Pode ser integrada em páginas ASP

# C# - Orientada a Objectos

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

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

# C# - Preservar Investimentos

- Semelhanças com C++ e Java
  - Espaços de nomes;
  - Nenhum sacrifício necessário
- Interoperabilidade
  - Cada vez mais importante
  - C# fala com XML, SOAP, COM, DLLs, e qualquer linguagem do .NET *FRAMEWORK*
- Milhões de linhas de código C# no .NET
  - Pequena curva de aprendizagem
  - Melhor produtividade



# C# - Tipos de Dados

- O C# é uma linguagem altamente “**Tipada**”, é necessário indicar qual o tipo de dados de cada objecto criado
- O compilador ajuda a prevenir erros forçando que unicamente a atribuição dos dados correctos ao objecto em questão.
- O tipo de um objecto indica ao compilador o tamanho do mesmo e as suas capacidades:
  - 1 **int** indica um objecto de 4 bytes,
  - Se for um **button**, pode ser pressionado, desenhado, etc.
- Tal como o C++ e o Java os tipos de dados dividem-se em dois conjuntos, intrínsecos e os definidos pelo programador.
- No entanto o C# ainda divide os conjuntos em duas outras categorias, **Tipos Valor** e **Tipos Referência**. A principal diferença reside na forma como são armazenados na memória

- **Tipos Valor**

- Variáveis contêm directamente dados/instâncias
  - Não podem ser **null**
  - Comparação e atribuição operam com os próprios valores (em C#)
  - Manipulação eficiente porque podem ser alocados na stack

- **Tipos Referência**

- Variáveis contêm referências para objectos/instâncias (***no heap***)
  - Podem ser **null**
  - Comparação e atribuição operam com referências
  - Gestão automática de memória (*garbage collection* do CLR)



- **Tipos Valor**

- Primitivos `int i;`
- Enumerações `enum State {On, OFF}`
- Estruturas `struct Point {int x,y;}`

- **Tipos Referência**

- Arrays `string[] a = new string[10];`
- Classes `class Foo:Bar,IFoo {...}`
- Interfaces `interface IFoo:IBar {...}`
- Delegados `delegate double MathFunc (double x);`

# C# - Tipos de Dados

CTS Type Name	Tamanho	.NET tipo	Descrição
System.Object		object	Classe base para todos os tipos CTS
System.String		string	String
System.SByte	1	sbyte	8 bits com sinal (-128 até 127)
System.Byte	1	byte	8bits sem sinal (0 até 255)
System.Int16	2	short	16 bits com sinal (-32,768 até 32767)
System.UInt16	2	ushort	16 bits sem sinal (0 até 65,535)
System.Int32	4	int	32 bits com sinal (-2,147,483,648 até 2,147,483,648)
System.UInt32	4	uint	32 bits sem sinal (0 até 4,294,967,295)
System.Int64	8	long	64 bits com sinal (-9,223,372,036,854,775,808 até 9,223,372,036,854,775,807)
System.UInt64	8	ulong	64 bits sem sinal (0 até 0xFFFFFFFFFFFFFFFF)
System.Char	2	char	Caracteres
System.Single	4	Float	Numeros de virgula Flutuante
System.Double	8	Double	Virgula flutuante de dupla precisão
System.Boolean	1	bool	True ou False
System.Decimal	16	decimal	Até 28 dígitos, normalmente utilizado em aplicações financeiras

- Uma **Stack** consiste numa estrutura para armazenar dados utilizando o conceito LIFO (last-in-First-out). A **Stack** refere-se a uma área de memória suportada pelo processador, na qual as variáveis locais são armazenadas.
- Na linguagem C#, os Tipo Valor (por exemplo inteiros) são alocados na **Stack**.
- Tipos Referência (por exemplo objectos) são armazenados na **Heap**. Quando um objecto é armazenado na **Heap**, o seu endereço é retornado e é atribuído a uma referência.
- O *garbage collector* destrói os objectos armazenados na **Stack**, após a estrutura desta ter sido eliminada. Tipicamente, uma estrutura de **Stack** é definida dentro de uma função. Sendo assim, todos os objectos declarados dentro da função, serão “marcados” para **garbage collection**, quando esta terminar a sua execução.
- Os objectos armazenados na **Heap**, são eliminados pelo *garbage collector*, após a sua referência ter sido eliminada.

# C# - Variáveis & Constantes

## Variáveis

- Uma variável é criada através da atribuição de um nome e da declaração do tipo.
- Pode ser inicializada quando se declara, e é possível alterar o seu valor em qualquer momento.
- O C# requiere atribuição de valores, isto é, uma variável não pode ser usada sem antes possuir um valor.

## Constantes

- Uma constante consiste numa variável cujo valor não pode ser modificado.
- As constantes podem ser de três tipos, *literais*, *constantes simbólicas* e *enumerações* (**enumerations**)

X = 32;                                      Literal

const int FreezingPoint = 32;                      constante simbólica



## Enumerations

- As enumerações constituem uma poderosa alternativa às constantes. Uma enumeração consiste num conjunto de constantes, denominada ***enumeration list***.
- Definição técnica de uma enumeração:

[attributes] [modifiers] enum identifier [:base-type] {enumerator-list};

```
enum ServingSizes :uint
{
    Small = 1,
    Regular = 2,
    Large = 3
}
```

# C# - Enumerações

```
class Values
{
    enum Temperatures: int
    {
        WickedCold =0,
        FreezingPoint = 32,
        LightJacketWeather = 60,
        SwimmingWeather = 72,
        BoilingPoint = 212,
    }
    static void Main(string[] args)
    {
        System.Console.WriteLine("Freezing point of water: {0}", (int) Temperatures.FreezingPoint);
        System.Console.WriteLine("Boiling point of water: {0}", (int) Temperatures.BoilingPoint);

        System.Console.Read();
    }
}
```

- É necessário especificar o tipo de dados que se pretende imprimir, caso seja omissso, o valor imprimido consiste no nome da constante.
- Todas as *enumerations list* possuem *scope*, isto permite possuir constantes com o mesmo nome em *enumerations list* diferentes.

# C# - Preprocessor Directives

- As directivas **#region** - **#endregion**, são usadas para indicar o início e o fim de um determinado bloco de código.

```
#region Funções de Teste  
    int x;  
    double d;  
    Currency balance;  
#endregion
```

## C# - Documentação XML

- A Linguagem C# permite gerar documentação em formato XML automaticamente a partir de comentários especiais no código.
- Esses comentários consistem em linhas únicas, iniciadas por `///` (3 barras).
- Dentro desses comentários colocam-se *tags* de XML que permitem gerar a documentação.



# C# - Documentação XML

TAG	Descrição
<c>	Especifica uma linha como sendo uma linha de código
<code>	Especifica várias linhas como sendo código
<example>	Especifica como sendo código exemplo
<exception>	Documenta uma exceção
<include>	Inclui comentários de outro ficheiro de documentação
<list>	Permite inserir uma lista dentro da documentação
<param>	Especifica 1 parâmetro do método.
<paramref>	Especifica que uma palavra é parâmetro do método.
<permission>	Documenta o acesso a um membro.
<remarks>	Adiciona uma descrição a um método.
<returns>	Documenta o valor de retorno do método.
<see>	Fornecer uma referência cruzada a um outro parâmetro.
<seealso>	Permite especificar um "ver também".
<summary>	Permite especificar um breve sumário.
<value>	Descreve uma propriedade

# C# - Documentação XML

```
using System;

namespace Math.Exemplo
{
    /// <summary>
    /// Classe de Matemática --> Exemplo para PSI
    /// Fornece um método para adicionar dois inteiros
    /// </summary>
    public class Math
    {
        ///<sumary>
        /// O método Add permite adicionar dois inteiros

        ///</sumary>
        ///<returns> O resultado da adicção é (int) </returns>
        ///<paran name="x"> Primeiro inteiro a adicionar </param>
        ///<paran name="y"> Segundo inteiro a adicionar </param>
        public int Add(int x, int y)
        {
            return x+y;
        }
    }
}
```



# C# - Operadores

- Um operador consiste num símbolo que obriga a linguagem C# a tomar uma acção.
- Existem diversos tipos de operadores:
  - Matemáticos,

Operador	Descrição
+	Adição
-	Subtracção
*	Multiplificação
/	Divisão
%	Resto da divisão

- Incrementos e Decrementos,
- Relacionais,
- Lógicos em operações condicionais.

# C# - Operadores

- Uma operação comum em programação é a necessidade de somar ou subtrair a uma variável, ou ainda, modificar o valor de uma variável, e atribuí-lo novamente à mesma variável.

Operador	Descrição
++	Incrementa um valor
--	Decrementa um valor
+=	Soma o valor à variável em questão
-=	Subtrai o valor à variável em questão
*=	Multiplifica o valor à variável em questão
/=	Divide o valor à variável em questão

```
myExample = 10;
myExample ++;           // myExample = 11
myExample += 9          // myExample = 20
myExample /= 5          // myExample = 4
myExample *= 10         // myExample = 40
```

# C# - Operadores

- Para complicar ainda mais as coisas, é possível incrementar uma variável e atribuir esse valor a outra variável.

```
firstvalue = secondvalue ++;
```

- Coloca-se uma questão com a atribuição anterior: o incremento é para ocorrer antes ou depois da atribuição?

## Sufixo

```
secondvalue = 10;
```

```
firstvalue = secondvalue++;    // firstvalue = 10 // secondvalue = 11
```

## Prefixo

```
secondvalue = 10;
```

```
firstvalue = ++secondvalue;    // firstvalue = 11 // secondvalue = 11
```

# C# - Operadores

- Operadores relacionais são utilizados para comparar dois valores, retorna como resultado um valor *booleano*.

Descrição	Operador	Expressão	Resultado
Igualdade	<code>==</code>	<code>bigvalue == 100</code>	<b>true</b>
		<code>bigvalue == 80</code>	<b>false</b>
Diferente	<code>!=</code>	<code>bigvalue != 100</code>	<b>false</b>
		<code>bigvalue != 80</code>	<b>true</b>
Maior que	<code>&gt;</code>	<code>bigvalue &gt; smallvalue</code>	<b>true</b>
Maior ou igual que	<code>&gt;=</code>	<code>bigvalue &gt;= smallvalue</code>	<b>false</b>
		<code>smallvalue &gt;= bigvalue</code>	<b>false</b>
Menor que	<code>&lt;</code>	<code>bigvalue &lt; smallvalue</code>	<b>false</b>
Menor ou igual que	<code>&lt;=</code>	<code>bigvalue &lt;= smallvalue</code>	<b>true</b>
		<code>smallvalue &lt;= bigvalue</code>	<b>false</b>

Assume-se como valores iniciais para as variáveis:

`bigvalue = 100` e `smallvalue = 50`

## Lógicos

Descrição	Operador	Expressão	Resultado
and	&&	(x==3) && (y == 7)	<b>false</b>
or		(x==3)    (y == 7)	<b>true</b>
not	!	! (x==3)	<b>true</b>

Assume-se como valores iniciais para as variáveis:

$x = 5$  e  $y = 7$

## typeof

- O operador **typeof** retorna o tipo de dados do objecto em questão.
- Isto torna-se útil quando se pretende usar reflexão de forma a obter informação sobre objecto criados dinamicamente.

## IS

- O operador **is** permite avaliar se um determinado objecto é compatível com um determinado tipo.

```
int i = 10;
if (i is object)
{
    Console.WriteLine("i é um objecto");
}
```

## AS

- O operador **as** é utilizado em conversões explícitas de tipos de dados. Se o tipo para o qual se pretende converter é compatível com o tipo de dados actual, a conversão é executada, caso contrário é retornado o valor **null**.

```
object o1 = "some string";
object o2 = 5;

string s1 = o1 as string;    // s1 = "some string"
string s2 = o2 as string;    // s2 = null
```



# C# - Operadores

## Precedência

Descrição	Operadores
Primário	() . x++ x- new typeof sizeof checked unchecked
Unários	+ - ! ~ ++x --x
Multiplicação/Divisão	* / %
Adição/subtração	+ -
Deslocamento de bits	<< >>
Relacionais	< > <= >= is as
Comparação	== !=
AND	&
XOR	^
OR	
Boolean AND	&&
Boolean OR	
Ternário	?:
Atribuição	= += -= *= /= %= &=  = ^= <<= >>=

# C# - Estruturas de Decisão

- **IF** - O mais simples dos comandos condicionais. Teste a condição e executa o comando, indicando se o resultado é TRUE. Para indicar mais do que um comando no IF, é necessário utilizar um bloco, demarcado por chavetas.

```
int i = 4;
if (i == 0)
{
    Console.WriteLine("i é zero");
}
else if (i==1)
{
    Console.WriteLine("i é um");
}
else
{
    Console.WriteLine("i é maior que um");
}
```

# C# - Estruturas de Decisão

- **SWITCH** - O comando SWITCH funciona como uma sequencia de comandos IF na mesma expressão. Permite ao programador comparar uma variável com uma conjunto de valores diferentes, e mediante isso executar um código diferente.

```
Switch (country)
{
    case "America":
        Console.WriteLine("O país em questão é a América");
        break;
    case "Inglaterra":
        Console.WriteLine("O país em questão é a Inglaterra");
        break;
    case "Portugal":
        Console.WriteLine("O país em questão é Portugal");
        break;
}
```

# C# - Estruturas de Decisão

```
Switch (country)
{
    case "US":
    case "UK":
    case "AU":
        language = "English";
        break;
    case "at":
    case "de":
        language = "German";
        break;
}
```

```
Switch (country)
{
    case "America":
        Console.WriteLine("O país em questão é a América");
        goto case "Inglaterra";
    case "Inglaterra":
        language = "English";
        break;
}
```

## C# - Estruturas de Ciclo

- **WHILE** - É o comando de repetição mais simples. Testa uma condição e executa um comando (ou conjunto de comandos) até a condição ser FALSE.

```
bool condition = false;
while (!condition)
{
    //fica dentro do ciclo até a condição ser true
    DoSomeWork();
    condition = checkCondition(); // assume checkCondition() retorna um bool
}
```

- **DO...WHILE** - Este comando funciona de forma bastante semelhante ao anterior, diferendo apenas no local onde a condição é testada. Neste comando a condição só é testada após a execução do bloco de comandos, obrigando à execução deste pelo menos uma vez.

```
bool condition;  
{  
    // executa pelo menos uma vez  
    DoSomeWork();  
    condition = checkCondition(); // assume checkCondition() retorna um bool  
} while (condition)
```

# C# - Estruturas de Ciclo

- **FOR** – é a estrutura de controlo mais complexa em C#. Nesta estrutura é necessário definir o valor de inicialização, a condição de paragem, e o valor de incremento.

```
For (int i = 0; i < 100; i++)  
{  
    Console.WriteLine(i);  
}
```

- **FOREACH** – permite interagir com os elementos de um colecção (por agora considera-se um objecto que possui outros objectos). É importante referir que dentro desta estrutura não é possível alterar o valor dos elementos da colecção.

```
foreach (int temp in arrayOfInts)  
{  
    Console.WriteLine(temp);  
    temp++;  
}
```

- A Linguagem C# trata os erros e condições anormais através de excepções.
- Uma excepção consiste num objecto que encapsula informação acerca de uma ocorrência fora do comum.
- É importante distinguir erros, *bugs* e excepções.
  - *Bug*: erro de programação que deverá ser corrigido.
  - Erro: acontece como consequência da acção do utilizador.
- Erros e *bugs* podem lançar uma excepção.
- É impossível prever todas as excepções, mas podem-se tratar de forma a evitar que o programa colapse.
- Quando uma excepção é lançada, a função actual pára, isto é, a função termina naquele momento, tendo no entanto a oportunidade de tratar a excepção. Se nenhuma das funções activas tratar a excepção, esta será tratada em última instância pelo CLR, o qual terminará o programa.



## throw

- Permite indicar a ocorrência de uma condição anormal.
- A execução de um programa é automaticamente parada ao ser lançada uma exceção, enquanto o CRL procura um “*handler*” que trate a exceção.
- Caso não encontre um *handler*, vai retornando nas funções até encontra algum que satisfaça a exceção. Caso retorne até ao método **main** e não encontre um *handler*, termina o programa.

## catch

- Um *handler* de uma exceção, é denominado por um bloco de **catch**.
- Um **catch** está associado a um bloco de dados que eventualmente poderá gerar um erro, esse código encontra-se dentro de um bloco **try**
- Os **catchs** podem ser gerais, ou específicos.

# C# - Exceções - throw

```
namespace MyThrow
{
    using System;
    public class Test
    {
        public static void Main()
        {
            Console.WriteLine
("Enter Main...");

            Test t = new Test();
            t.Func1();
            Console.WriteLine
("Exit Main...");
        }
        public void Func1()
        {
            Console.WriteLine
("Enter Func1...");

            Func2();
            Console.WriteLine
("Exit Func1...");
        }
        public void Func2()
        {
            Console.WriteLine
("Enter Func2...");

            throw new System.Exception();
            Console.WriteLine
("Exit Func2...");
        }
    }
}
```

## OUTPUT:

```
Enter Main...
Enter Func1...
Enter Func2...
```

Exception occurred: System.Exception: Na exception of type System.Exception was throw.

```
at MyThrow.Test.Func2();
    in ...exception01.css:line 23

at MyThrow.Test.Func1();
    in ...exception01.css:line 17

at MyThrow.Test.Main();
    in ...exception01.css:line 11
```



# C# - Exceções - catch

```
namespace MyThrow {
    using System;
    public class Test    {

        public static void Main() {
            Console.WriteLine("Enter Main...");
            Test t = new Test();
            t.Func1();
            Console.WriteLine("Exit Main...");
        }
        public void Func1() {
            Console.WriteLine("Enter Func1...");
            Func2();
            Console.WriteLine("Exit Func1...");
        }

        public void Func2() {
            try {
                Console.WriteLine("Enter Func2...");
                throw new System.Exception();
                Console.WriteLine("Exit try Block...");
            }
            catch
            {
                Console.WriteLine("Exception caught and
handled");
            }
            Console.WriteLine("Exit Func2...");
        }
    }
}
```

## OUTPUT:

```
Enter Main...
Enter Func1...
Enter Func2...
Entering try Block
Exception caught and handled.
Exit Func2...
Exit Func1...
Exit Main...
```

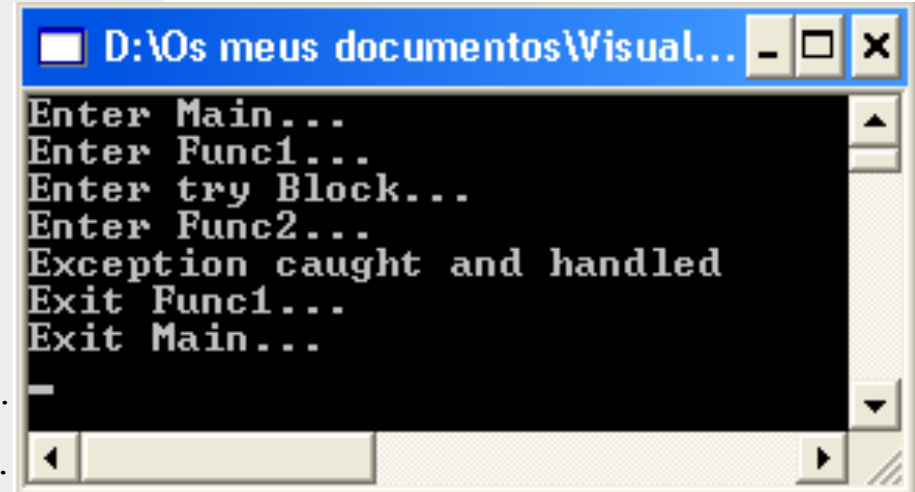


## C# - Exceções – catch – exemplo 2

```
namespace MyThrow {
    using System;
    public class Test {

        public static void Main() {
            Console.WriteLine("Enter Main...");
            Test t = new Test();
            t.Func1();
            Console.WriteLine("Exit Main...");
        }
        public void Func1() {
            Console.WriteLine("Enter Func1...");
            try {
                Console.WriteLine("Enter try Block..");
                Func2();
                Console.WriteLine("Exit try Block..");
            }
            catch{
                Console.WriteLine("Exception caught and
handled");
            }
            Console.WriteLine("Exit Func1...");
        }

        public void Func2() {
            Console.WriteLine("Enter Func2...");
            throw new System.Exception();
            Console.WriteLine("Exit Func2...");
        }
    }
}
```



```
D:\Os meus documentos\Visual...
Enter Main...
Enter Func1...
Enter try Block...
Enter Func2...
Exception caught and handled
Exit Func1...
Exit Main...
-
```

## C# - Exceções – catch dedicados

```
namespace MyDedicatedCatch
{
    using System;
    public class DedicatedCatch
    {
        public static void Main()
        {
            DedicatedCatch t = new DedicatedCatch();
            t.TestFunc();
            Console.ReadLine();
        }

        //do the division if legal
        public double DoDivide(double num1, double num2)
        {
            if (num2==0)
                throw new System.DivideByZeroException();
            if (num1==0)
                throw new System.ArithmeticException();

            return num1/num2;
        }
    }
}
```

## C# - Exceções – catch dedicados 2

```
// Try to divide 2 numbers handle possible exceptions
public void TestFunc()
{
    try{
        double a = 5;
        double b = 0;
        Console.WriteLine("{0}/{1} = {2}", a,b,DoDivide(a,b));
    }
    //most derived exception type first
    catch (System.DivideByZeroException){
        Console.WriteLine("DivideByZeroException caught!");
    }
    catch (System.ArithmeticException){
        Console.WriteLine("ArithmeticException caught!");
    }
    //generic exception type last
    catch {
        Console.WriteLine("Unknow exception caught!");
    }
}
}
```

## finally

- Nalgumas situações lançar uma exceção pode criar problemas graves. Por exemplo se existe um ficheiro aberto, ou se existe a pertença de um recurso, é importante ter oportunidade de fechar o ficheiro, ou limpar o *buffer*.
- Existe sempre a possibilidade de dentro de um bloco **catch** resolver o problema, mas isso implicava repetição de código (Exemplo: fechar o ficheiro dentro do bloco **try**, e precaver fechando também dentro do bloco **catch**)
- O bloco **finally** garante a execução de código apesar da ocorrência de uma exceção.
- Um bloco **finally** pode existir sem um bloco **catch**, mas necessita obrigatoriamente de um bloco **try**.
- É um erro sair de um bloco **finally** com *break*, *continue*, *return* ou *goto*.

# C# - Excepções – finally

```
namespace MyDedicatedCatch
{
    using System;
    public class DedicatedCatch
    {
        public static void Main()
        {
            DedicatedCatch t = new DedicatedCatch();
            t.TestFunc();
            Console.ReadLine();
        }

        //do the division if legal
        public double DoDivide(double num1, double num2)
        {
            if (num2==0)
                throw new System.DivideByZeroException();
            if (num1==0)
                throw new System.ArithmeticException();

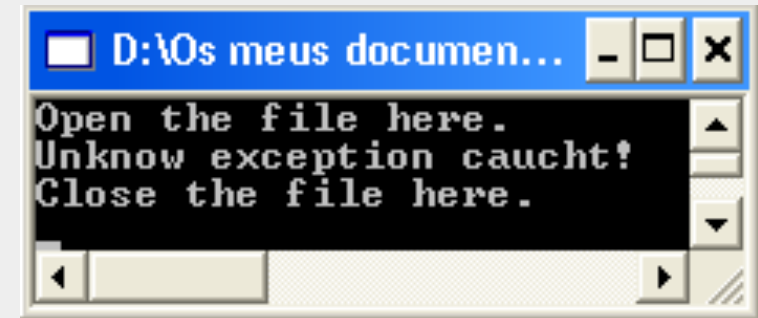
            return num1/num2;
        }
    }
}
```





# C# - Exceções – finally 2

```
// Try to divide 2 numbers handle possible exceptions
public void TestFunc()
{
    try{
        Console.WriteLine("Open the file here.");
        double a = 5;
        double b = 0;
        Console.WriteLine("{0}/{1} = {2}", a,b,DoDivide(a,b));
    }
    //most derived exception type first
    catch (System.DivideByZeroException){
        Console.WriteLine("DivideByZeroException caught!");
    }
    //generic exception type last
    catch {
        Console.WriteLine("Unknow exception caught!");
    }
    finally
    {
        Console.WriteLine("Close the file here.");
    }
}
```



## C# - Objecto Excepção

- O objecto `System.Exception` fornece um número de métodos e propriedades que permitem obter mais informação sobre a excepção e suas causas.
- A propriedade `Message` permite obter informação acerca da excepção e razão de ter ocorrido. É *read-only*.
- A propriedade `HelpLink` fornece um *link* de ajuda com informação acerca da excepção. É *read/write*.
- A propriedade `StackTrace` é *read-only* e é activada em tempo de execução, fornece informação acerca do erro ocorrido.

# C# - Objecto Excepção

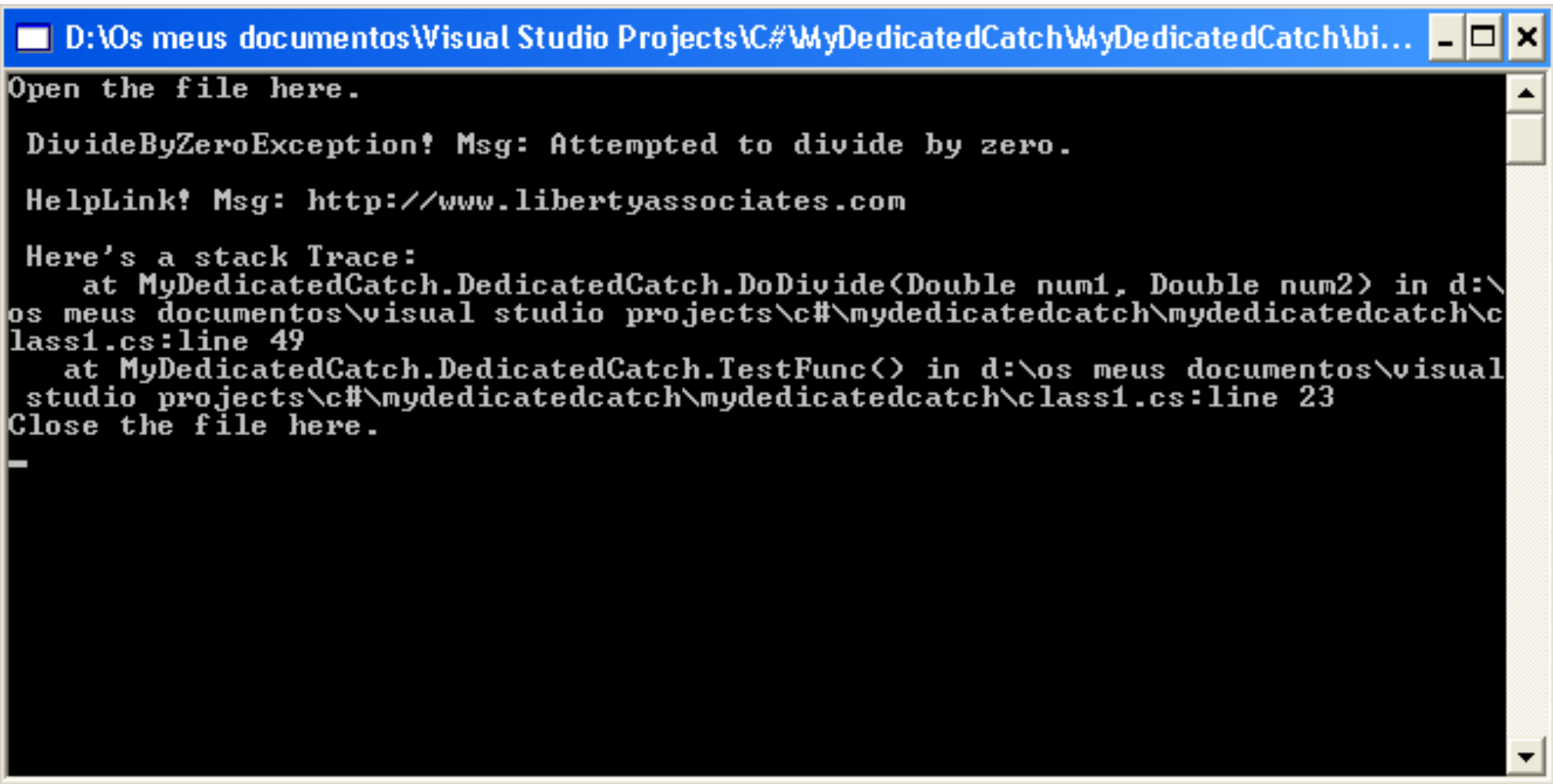
```
namespace MyDedicatedCatch
{
    using System;
    public class DedicatedCatch
    {
        public static void Main()
        {
            DedicatedCatch t = new DedicatedCatch();
            t.TestFunc();
            Console.ReadLine();
        }
        //do the division if legal
        public double DoDivide(double num1, double num2)
        {
            if (num2==0){
                DivideByZeroException e = new DivideByZeroException();
                e.HelpLink = "http://www.libertyassociates.com";
                throw e;
            }
            if (num1==0)
                throw new System.ArithmeticException();

            return num1/num2;
        }
    }
}
```

## C# - Objecto Excepção 2

```
// Try to divide 2 numbers handle possible exceptions
public void TestFunc()
{
    try {
        Console.WriteLine("Open the file here.");
        double a = 5;
        double b = 0;
        Console.WriteLine("{0}/{1} = {2}", a,b,DoDivide(a,b));
    }
    //most derived exception type first
    catch (System.DivideByZeroException e) {
        Console.WriteLine("\n DivideByZeroException! Msg: {0}", e.Message);
        Console.WriteLine("\n HelpLink! Msg: {0}", e.HelpLink);
        Console.WriteLine("\n Here's a stack Trace: {0}", e.StackTrace);
    }
    //generic exception type last
    catch {
        Console.WriteLine("Unknow exception caught!");
    }
    finally
    {
        Console.WriteLine("Close the file here.");
    }
}
}
```

## C# - Objecto Excepção 3



```
D:\Os meus documentos\Visual Studio Projects\C#\MyDedicatedCatch\MyDedicatedCatch\bi...
Open the file here.
DivideByZeroException! Msg: Attempted to divide by zero.
HelpLink! Msg: http://www.libertyassociates.com
Here's a stack Trace:
    at MyDedicatedCatch.DedicatedCatch.DoDivide(Double num1, Double num2) in d:\os meus documentos\visual studio projects\c#\mydedicatedcatch\mydedicatedcatch\class1.cs:line 49
    at MyDedicatedCatch.DedicatedCatch.TestFunc() in d:\os meus documentos\visual studio projects\c#\mydedicatedcatch\mydedicatedcatch\class1.cs:line 23
Close the file here.
```

# C# - Classes e Objectos

- A diferença entre classe e objecto é a mesma que entre o conceito de cão e um cão específico. Uma classe cão descreve o cão em si: a cor, a raça, o peso, o tamanho, etc. Também descreve as acções que o cão pode tomar: comer, ladrar, andar, dormir. Um cão específico tem características específicas, 10 quilos, olhos pretos, raça Dálmata.
- A grande vantagem da programação orientada aos objectos é o encapsulamento das características e capacidades de uma entidade num único bloco de código.
- Encapsulamento, polimorfismo e herança, são os três pilares base da programação orientada a objectos.
- Ao definir uma nova classe é necessário primeiro declará-la, e so depois declarar os métodos e os campos.

*[attributes] [access-modifiers] class identifier [:base-class]  
{class-body}*



# C# - Classes e Objectos

- Um classe é definida dentro de um chavetas: {}
- Normalmente o access-modifier utilizado é o **public**

```
Public class Tester
{
    public static int Main()
    {
        //.....
    }
}
```

- Ao declarar uma nova classe, definem-se as propriedades dos objectos das classes assim como o seu comportamento.
- Só existem objectos da classe no fim de instanciados.

# C# - Classes e Objectos

- *access-modifiers* determinam quais os métodos da classe (incluindo métodos das outras classes) que podem visualizar e aceder a variáveis ou métodos desta classe.

<b><i>Access-modifiers</i></b>	<b>Restrições</b>
public	Sem restrições, membros definidos como <b>public</b> são visíveis a qualquer método de qualquer classe.
private	Membros na classe A definidos como <b>private</b> são acessíveis unicamente a métodos desta classe.
protected	Membros na classe A definidos como <b>protected</b> são acessíveis a métodos da classe A e a métodos de classes derivadas desta.
internal	Membros na classe A definidos como <b>internal</b> são acessíveis unicamente a métodos da classe A durante a <i>assemblagem</i> .
protected internal	Membros na classe A definidos como <b>protected internal</b> são acessíveis a métodos da classe A e a métodos de classes derivadas desta durante a <i>assemblagem</i> . O conceito é <b>protected</b> OU <b>internal</b> .



## C# - Classes e Objectos

- É preferível definir as variáveis membro da classe como **private**. Isto implica que o acesso aos valores das variáveis seja efectuado somente com recurso a métodos da classe.
- Por defeito a linguagem C# especifica tudo como **private**, pelo que não é necessário indicar explicitamente, no entanto é recomendável.

```
// private variables  
  
private int Year;  
private int Month;  
private int Date;  
private int Hour;  
private int Minute;  
private int Second;
```

# C# - Classes - Construtor

- O construtor de uma classe consiste no método que é invocado aquando a instanciação de um objecto.
- Pode ser omitido, e deixar o CLR fornecer um construtor por defeito.
- O objectivo do construtor é criar um objecto do tipo da classe e colocá-lo num estado válido.

Tipo	Valor por defeito
numeric (int, long, etc)	0
bool	False
char	'\0' (null)
enum	0
reference	null

# C# - Classes – Construtor

```
public class Time
{
    //private member variables
    int Year;
    int Month;
    int Date;
    int Hour;
    int Minute;
    int Second;

    // public acessor Methods
    public void DisplayCurrentTime()
    {
        System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
            Month, Date, Year, Hour, Minute, Second);
    }

    //constructor
    public Time(System.DateTime dt)
    {
        Year = dt.Year;
        Month = dt.Month;
        Date = dt.Day;
        Hour = dt.Hour;
        Minute = dt.Minute;
        Second = dt.Second;
    }
}
```



## C# - Classes – Construtor 2

```
public class Tester
{
    static void Main()
    {
        System.DateTime currentTime = System.DateTime.Now;
        Time t = new Time(currentTime);
        t.DisplayCurrentTime();
        System.Console.ReadLine();
    }
}
```

- A palavra reservada **this** refere-se à instância actual de um objecto.
- A referência **this** representa 1 ponteiro para todos os métodos **não static** da classe.
- Existem 3 formas de usar o **this** (só serão analisadas 2):
  - Para atribuição de valores a variáveis da classe:

```
public void SomeMethod (int Hour)
{
    this.Hour = Hour;
}
```

- De forma a permitir passar o objecto actual como parâmetro a outro método:

```
public void FirstMethod (OtherClass otherObject)
{
    otherObject.SecondMethod(this);
}
```

## C# - Classes – Copy

- O construtor *copy* cria um novo objecto através da copia do conteúdo das variáveis de um objecto existente do mesmo tipo.
- C# não fornece nenhum construtor *copy*, pelo que caso seja necessário, é preciso implementá-lo.

```
public Time(Time existingTimeObject)
{
    this.Year = existingTimeObject.Year;
    this.Month = existingTimeObject.Month;
    this.Date = existingTimeObject.Date;
    this.Hour = existingTimeObject.Hour;
    this.Minute = existingTimeObject.Minute;
    this.Second = existingTimeObject.Second;
}
```

# C# - Membros Estáticos

- Membros estáticos são considerados parte da classe.
- Para aceder a um membro estático, é necessário indicar primeiro o nome da classe a que pertence.
- Suponhamos que existe uma classe denominada Button que possui objectos instanciados (btnUpdate e btnDelete). Suponhamos ainda que a classe possui um método **static** denominado *SomeMethod()*. Acede-se ao método da seguinte forma:

```
Button.SomeMethod();
```

- E não:

```
btnUpdate.SomeMethod();
```

- Em C# não é permitido aceder a um método ou membro **static** através de uma instância da classe. Dá erro de compilação.

## C# - Membros Estáticos

- Não é possível saber com exactidão o construtor **static** será executado, sabe-se somente que ele será executado após o início do programa e antes de ser criada a primeira instância.
- Os membros estáticos são normalmente utilizados como contador de instâncias, isto é, através deles é possível ter conhecimento de quantas instâncias da classe existem.
- Os membros **static** são considerados parte da classe pelo que não podem ser inicializados numa instância. Sendo assim, requerem uma inicialização aquando da sua declaração.
- Os membros **static** não aceitam *access-modifier* (e.g. **public**)



# C# - Membros Estáticos

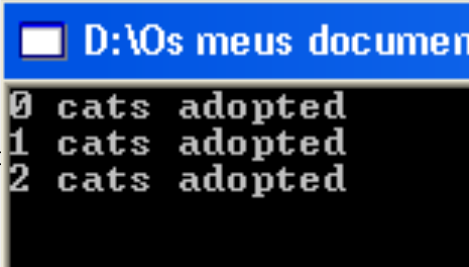
```
namespace MyStatic
{
    using System;

    public class Cat
    {
        private static int instances = 0;

        public Cat()
        {
            instances++;
        }

        public static void HowManyCats()
        {
            Console.WriteLine("{0} cats adopted", instances);
        }
    }

    public class Tester
    {
        static void Main()
        {
            Cat.HowManyCats();
            Cat Frisky = new Cat();
            Cat.HowManyCats();
            Cat Whiskers = new Cat();
            Cat.HowManyCats();
            Console.ReadLine();
        }
    }
}
```



D:\Os meus documentos

0	cats	adopted
1	cats	adopted
2	cats	adopted

## C# - Passagem de Parâmetros

- Por defeito a passagem de parâmetros para os métodos é efectuada por valor. Isto significa que quando o valor de um objecto é passado a um método, é criada uma cópia temporária desse objecto dentro do método. Uma vez executado o método, a copia é eliminada.
- É possível passar parâmetros por referência, usando **ref**.
- Os métodos só retornam um valor. (No entanto podem retornar uma colecção de valores).

# C# - Passagem de Parâmetros

```
namespace MyConstrutor
{
    public class Time
    {
        //private member variables
        int Year;
        int Month;
        int Date;
        int Hour;
        int Minute;
        int Second;

        // public accessor Methods
        public void DisplayCurrentTime()
        {
            System.Console.WriteLine("{0}/{1}/{2} {3}:{4}:{5}",
                                     Minute, Second);
        }

        public int GetHour()
        {
            return Hour;
        }

        public void GetTime(ref int h, ref int m, ref int s)
        {
            h = Hour;
            m = Minute;
            s = Second;
        }
    }
}
```



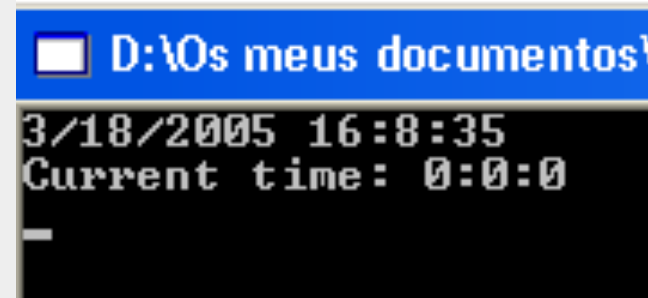
# C# - Passagem de Parâmetros 2

```
//constructor
public Time(System.DateTime dt)
{
    Year = dt.Year;
    Month = dt.Month;
    Date = dt.Day;
    Hour = dt.Hour;
    Minute = dt.Minute;
    Second = dt.Second;
}

public class Tester
{
    static void Main()
    {
        System.DateTime currentTime = System.DateTime.Now;
        Time t = new Time(currentTime);
        t.DisplayCurrentTime();

        int theHour = 0;
        int theMinute = 0;
        int theSecond = 0;

        t.GetTime(ref theHour, ref theMinute, ref theSecond);
        System.Console.WriteLine("Current time: {0}:{1}:{2}",
                                theHour, theMinute, theSecond);
        System.Console.ReadLine();
    }
}
```



## C# - Passagem de Parâmetros

- Como os inteiros são tipos valor, são passados como valor, é efectuada uma copia no método GetTime().
- De forma a tornar o método válido, é necessário efectuar passagem de parâmetros por referência.

```
public void GetTime(ref int h,ref int m,ref int s)
{
    h = Hour;
    m = Minute;
    s = Second;
}
```

```
// chamada à função.
t.GetTime(ref theHour,ref theMinute,ref theSecond);
```

# C# - Passagem de Parâmetros

- C# impõe que todas as variáveis seja inicializadas antes de ser utilizadas. É essa a razão da inicialização das variáveis theHour, theMinute e the Second a zero, antes de serem utilizadas.
- Caso não fossem inicializadas o compilador indicaria o seguinte erro:  
Use of unassignedlocal variable 'theHour'  
Use of unassignedlocal variable 'theMinute'  
Use of unassignedlocal variable 'theSecond'
- De forma a evitar este tipo de situações, a linguagem C# fornece um modificador denominado **out**, que permite passar uma variável a um método sem necessidade de inicialização.

```
public void GetTime(out int h, out int m, out int s)
{
    h = Hour;
    m = Minute;
    s = Second;
}

// chamada à função.
t.GetTime(out theHour, out theMinute, out theSecond);
```

# C# - Sobrecarga - Overloading

- Sobrecarga acontece quando se pretende implementar mais do que um método com o mesmo nome.
- O exemplo mais comum da utilização de sobrecarga é no método construtor.
- Um método é diferenciado por dois aspectos, o nome e os parâmetros.

```
void myMethod(int p1);
```

```
void myMethod(int p1, int p2);
```

```
void myMethod(int p1, string p2);
```

- Modificar o retorno de um método (deixando o nome e os parâmetros iguais) não permite sobrecarga, e o compilador indicará um erro.

# C# - Encapsulamento de dados

- Propriedades permitem aos clientes aceder ao estado da classe como se estivessem a aceder aos membros desta directamente.
- Propriedades têm dois objectivos:
  - Fornecem um interface simples para com o cliente, simulando uma variável membro.
  - São implementadas como métodos, no entanto, garantem o encapsulamento dos dados, respeitando as boas práticas de programação orientada a objectos.

```
public int Hour
{
    get
    {
        return Hour;
    }
    set
    {
        Hour = value;
    }
}
```



# C# - Encapsulamento de dados

## GET

- O bloco GET é semelhante a um método da classe que retorna um objecto do tipo da propriedade.
- Sempre que existir uma referência à propriedade (desde que não seja uma atribuição), o método GET é invocado para ler o valor da propriedade.

```
Time t = new Time(currentTime);  
  
int theHour = t.Hour;
```

## SET

- O bloco SET permite a atribuição de um valor e é semelhante a um método da classe que retorna **void**.
- Sempre que se executar uma atribuição de valor à propriedade, o método SET é invocado e o parâmetro implícito **value** toma o valor que se pretende atribuir.

```
theHour++;  
  
t.Hour = theHour;
```