
Programação Web

Aulas Teóricas – Capítulo 1 – 1.1

1º Semestre - 2023/2024

Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra



Programação Web

Introdução ao C# – Conceitos Básicos

Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra



C# vs ASP.Net Core

- **C#** é uma linguagem de programação
 - Actualmente está na Versão 11 (Novembro de 2022)
- O **ASP.Net Core** é um *framework* para desenvolvimento de aplicações para a *Web* e suporta várias linguagens de programação:
 - C#
 - F#
 - VB.Net (parcialmente)...

.Net

- **CLR (Common Language Runtime)**



Codificar neste
ambiente...



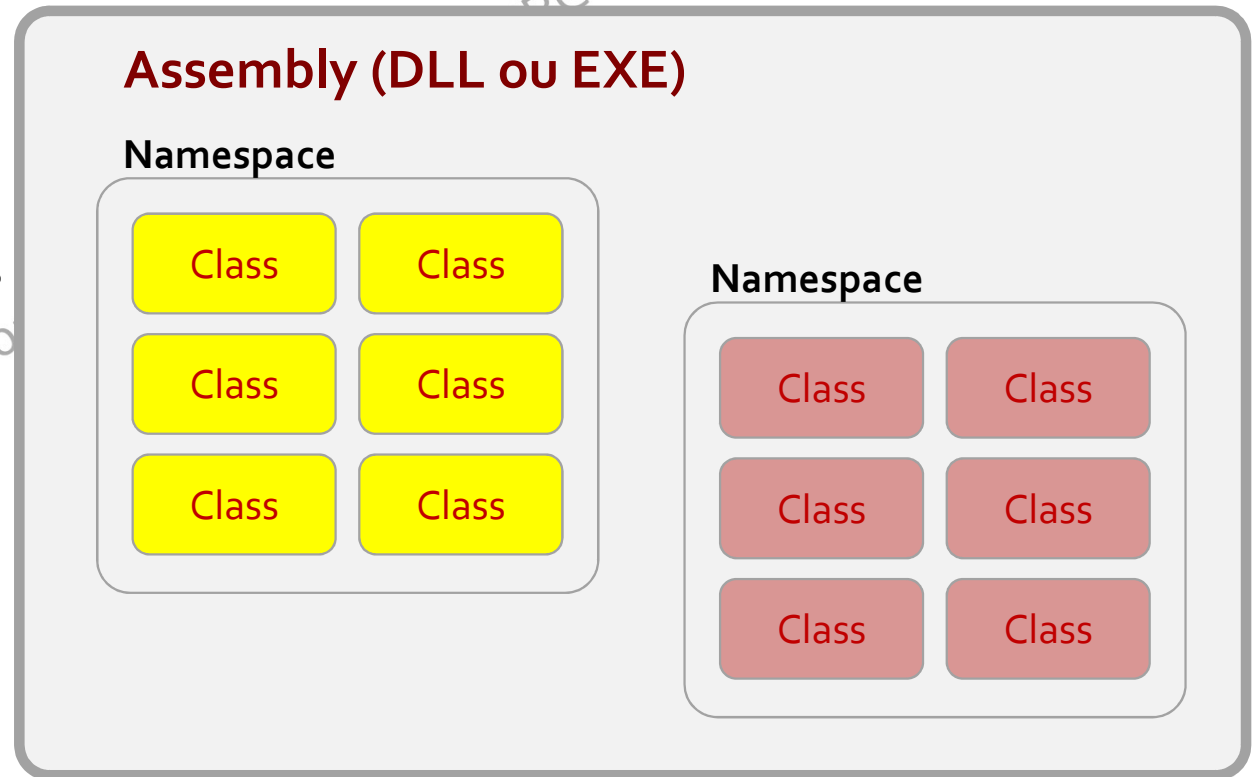
Executar neste....

- Conversão da Linguagem Intermédia (*IL Code*) para código máquina - *Just-in-time(JIT) compilation*.
- Integra serviços adicionais como gestão de memória, tratamento de excepções, *garbage collector*, segurança, gestão de *thread*,...
- **CLS, CTS ...**
- *Class Library*

Arquitetura de aplicações .NET

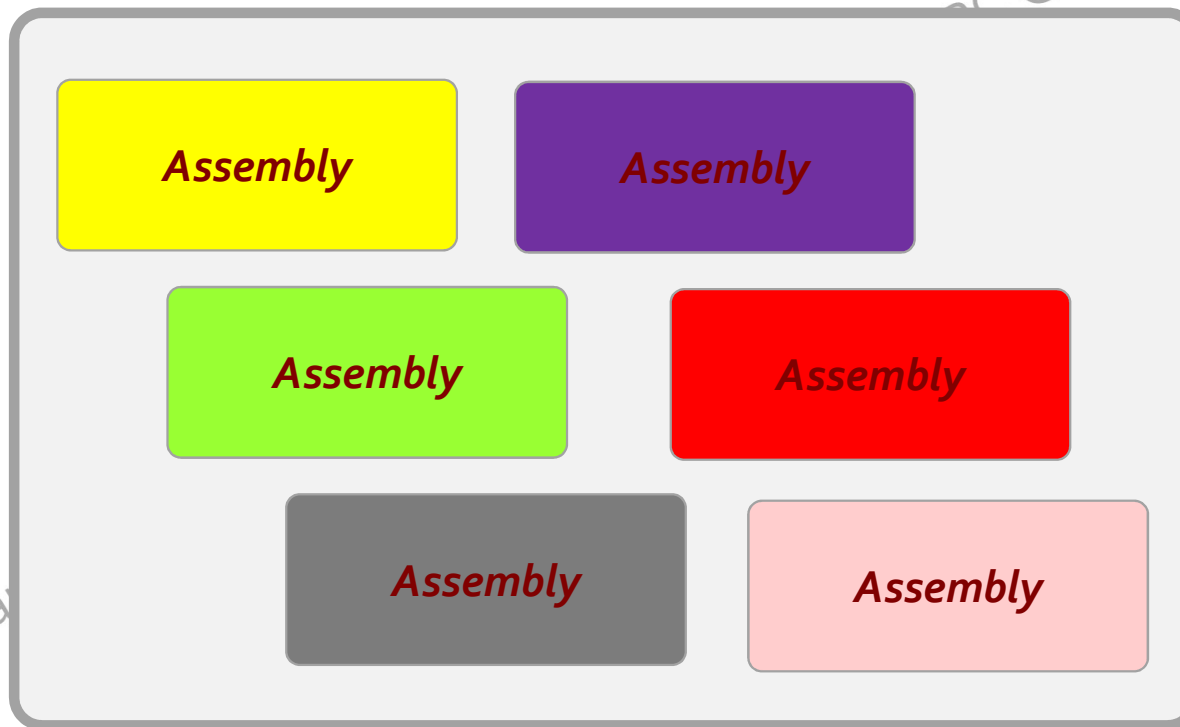
- **Namespaces** permitem organizar as classes

- Bases de dados
- Trabalhar com gráficos e imagens
- Segurança
- ...

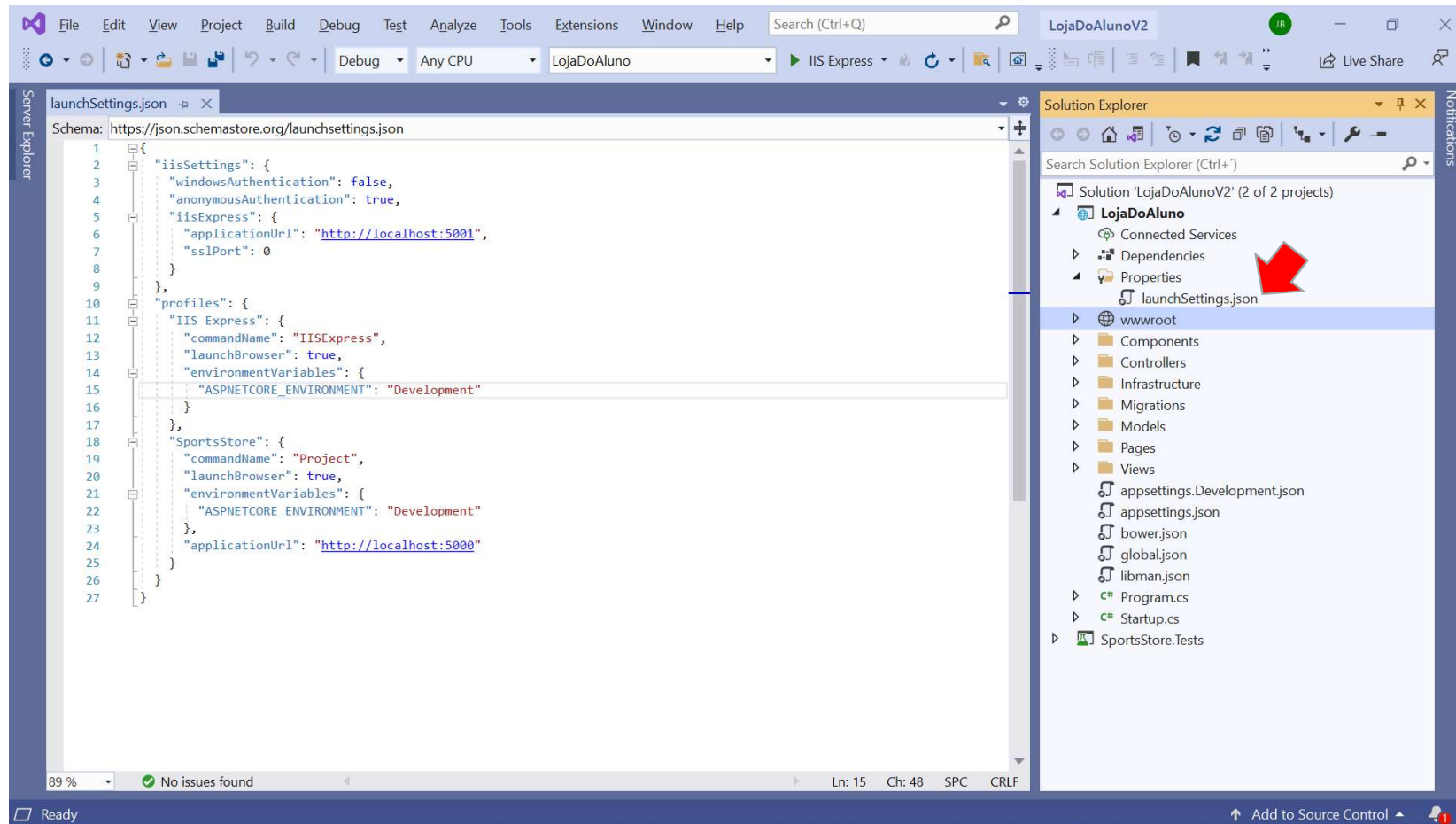


Arquitetura de aplicações .NET

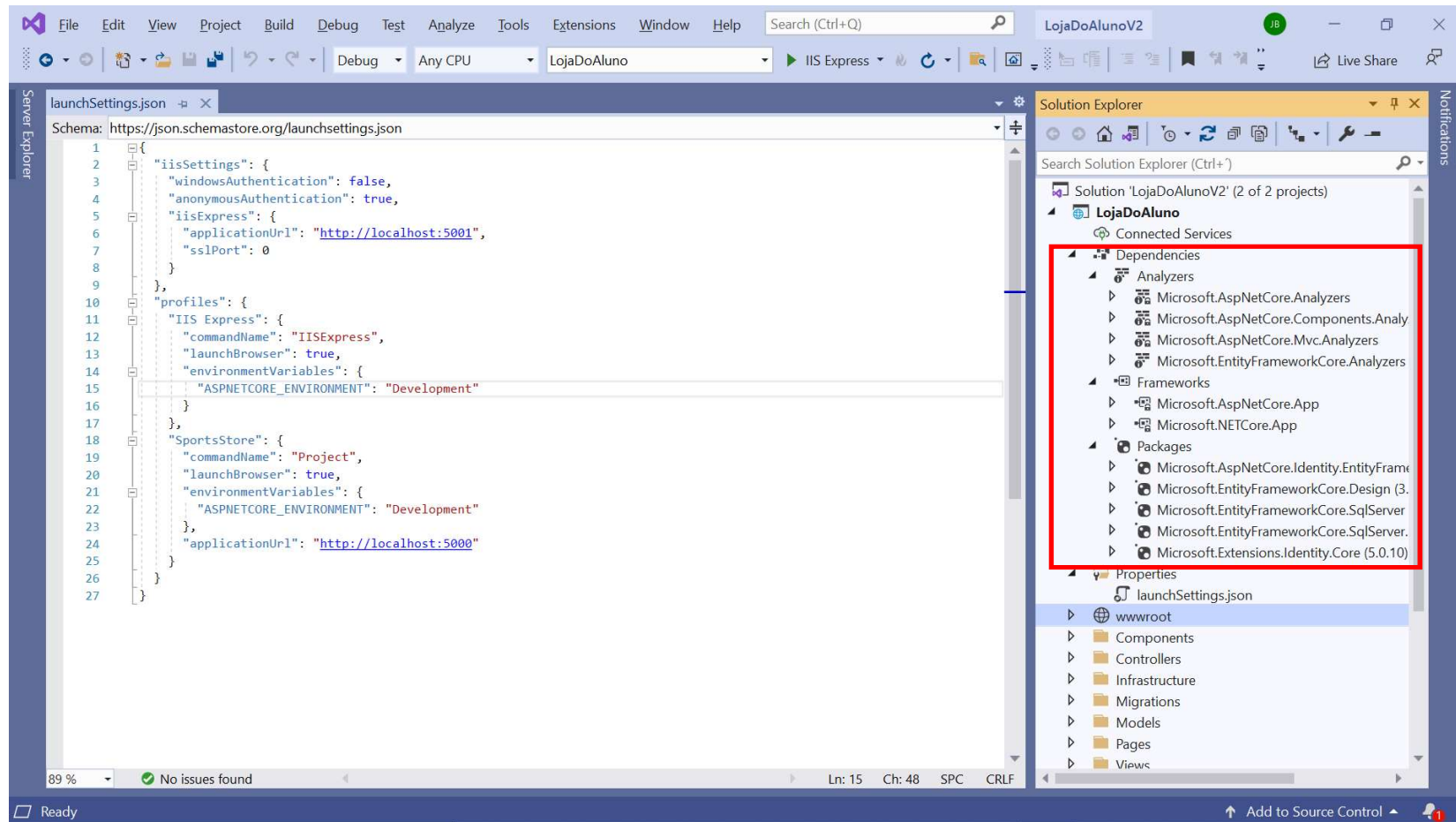
Aplicação



Arquitetura de aplicações ASP.Net Core



Arquitetura de aplicações ASP.Net Core




Introdução ao C#

- Desenvolvida pela Microsoft
- Baseada no C (Mistura de C++ e Java)
 - Orientada a Objetos
- Principal linguagem de programação para *framework* .NET
- Criador: Anders Hejlsberg
- Introduzido C#1.0 com VS2002, atualmente vai na versão C# 11.0 com VS2022 (ASP.Net Core 6)
- Possui um elevado nível de abstração



Introdução ao C#

C#	
	
Paradigma	estruturada · imperativa · concorrente · funcional · genérica · orientada a eventos · orientada a objetos · reflexiva
Surgido em	julho de 2000 (21 anos)
Última versão	9.0 (10 de novembro de 2020: há 10 meses ^[1])
Criado por	Microsoft
Estilo de tipagem	estática · dinâmica · forte · segura · insegura · nominativa · inferida
Principais implementações	.NET Framework · Mono · DotGNU · .Net Core
Dialetos:	Cw
Influenciada por	Java ^[2] · C++ · Eiffel · Modula-3 · Object Pascal ^[3] · Rust · F# · Haskell
Influenciou	D · Fantom · Java 5 ^[4] · Nemerle · Vala · Dart · Rust · F# · Swift · Hack
Licença:	MIT
Extensão do arquivo:	.cs
Página oficial	msdn.microsoft.com

+PL – ISEC/IPC @JB

Características Gerais

- Todos os ficheiros C# tem extensão .cs
- C# é *case sensitive*

```
Console.WriteLine("Programação em C#");
```

```
Console.Writeline("Programação em C#");
```

```
console.WriteLine("Programação em C#");
```

- O compilador C# ignora espaços adicionais
 - Sequencia de espaços, caracteres das tabs, *carriage return*, ...

Características Gerais

- Cada *statement* C# deve terminar com um ponto e vírgula (;)
 - Podem existir vários *statements* numa só linha
 - Por questões de legibilidade, muda-se de linha no final de cada *statement*
- Um bloco de código C# está delimitado por { e }
 - Pode conter qualquer número de linhas de código ou nenhum
 - Estes não necessitam de ser acompanhados pelo ‘;’

Características Gerais

- Comentários

- Comentar uma só linha: `// ...`
- Comentar mais do que uma linha: `/*
..... */`

```
int x; // Isto é um comentário
```

```
/* Isto..  
.....  
...Também é um comentário! */
```

Estrutura Básica de um programa em C#

System é um namespace

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;
```

Classes organizadas em *namespaces*

Console é uma classe do namespace System

```
namespace CSharpExampleConsole  
{  
    0 references  
    class Program  
    {  
        0 references  
        static void Main(string[] args)  
        {  
            Console.WriteLine("Hello World!");  
        }  
    }  
}
```

Estrutura Básica de um programa em C#

```
using System;  
using System.Collections.Generic;  
using  
using  
using
```

```
namespace  
{  
    or  
    cl  
{  
  
static void Main(string[] args)  
{  
    Console.WriteLine("Hello world!");  
}  
}  
}
```

- Contém unicamente um método chamado *Main*.
- *Main* contém uma linha de código que escreve “Hello, World!”
- O método que efetua esta ação chama-se *WriteLine*
- O método *WriteLine* pertence ao objeto *System.Console*
- A palavra chave “*static*” especifica que o método *Main* pode ser chamado mesmo que não haja uma instância da classe.

```
static void Main(string[] args)  
{  
    Console.WriteLine("Hello world!");  
}
```

Características Gerais

- #region
 - Permite comprimir/expandir o código de forma a ocultar/apresentar os detalhes de um bloco

```
#region Using Diretivas
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
#endregion
```

Programação Web

C#: Variáveis

Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra



C#: Literais

■ *Reais*

- Usado para valores do tipo ***float***, ***double*** e ***decimal***
- Podem ter notação exponencial
- O sufixo 'f' ou 'F' significa ***float***
- O sufixo 'd' ou 'D' significa ***double***
- O sufixo 'm' ou 'M' significa ***decimal***
- A interpretação por omissão é double

```
float realNumber = 12.5f;
```

C#: Literais - *Escape* caracteres

■ *String*

Sequência de <i>Escape</i>	Carater produzido
\'	Pelica
\"	Aspas
\\	Barra invertida
\0	Null
\a	Alert (origina um <i>beep</i>)
\b	Backspace
\f	<i>Form feed</i>
\n	Mudança de linha
\r	<i>Carriage return</i>
\t	Tabulação horizontal
\v	Tabulação vertical

Programação Web –

C#: Literais

■ **String** (cont.)

- Pode conter o prefixo @ (*verbatim strings literals*), permitindo reduzir as sequências de *escape*

```
string citacao= "\"Caros, Aula de PW!\", .....";  
string caminho = "C:\\3Ano\\PW\\exemplo.exe";  
citacao = @"""Caros, Aula de PW!""", .....";  
caminho = @"C:\3Ano\PW\exemplo.exe";  
string str = @"string exemplo";
```

Verbatim strings

Tipos de Dados em C#: Exemplo

```
int meuInteiro;  
string minhaString;
```

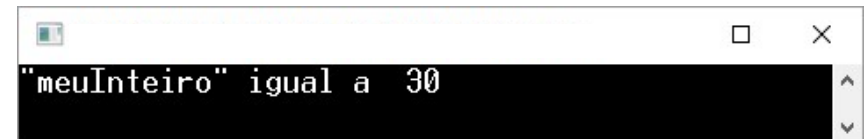
Atribuição de
valores fixos
Valores literais

```
meuInteiro = 30;  
minhaString = "\"meuInteiro\" igual a ";
```

Escape
carateres

```
Console.WriteLine($"{minhaString} {meuInteiro}");  
Console.ReadKey();
```

Característica do C#6
String Interpolation



```
"meuInteiro" igual a 30
```

Tipos de Dados em C#: Dinâmicos

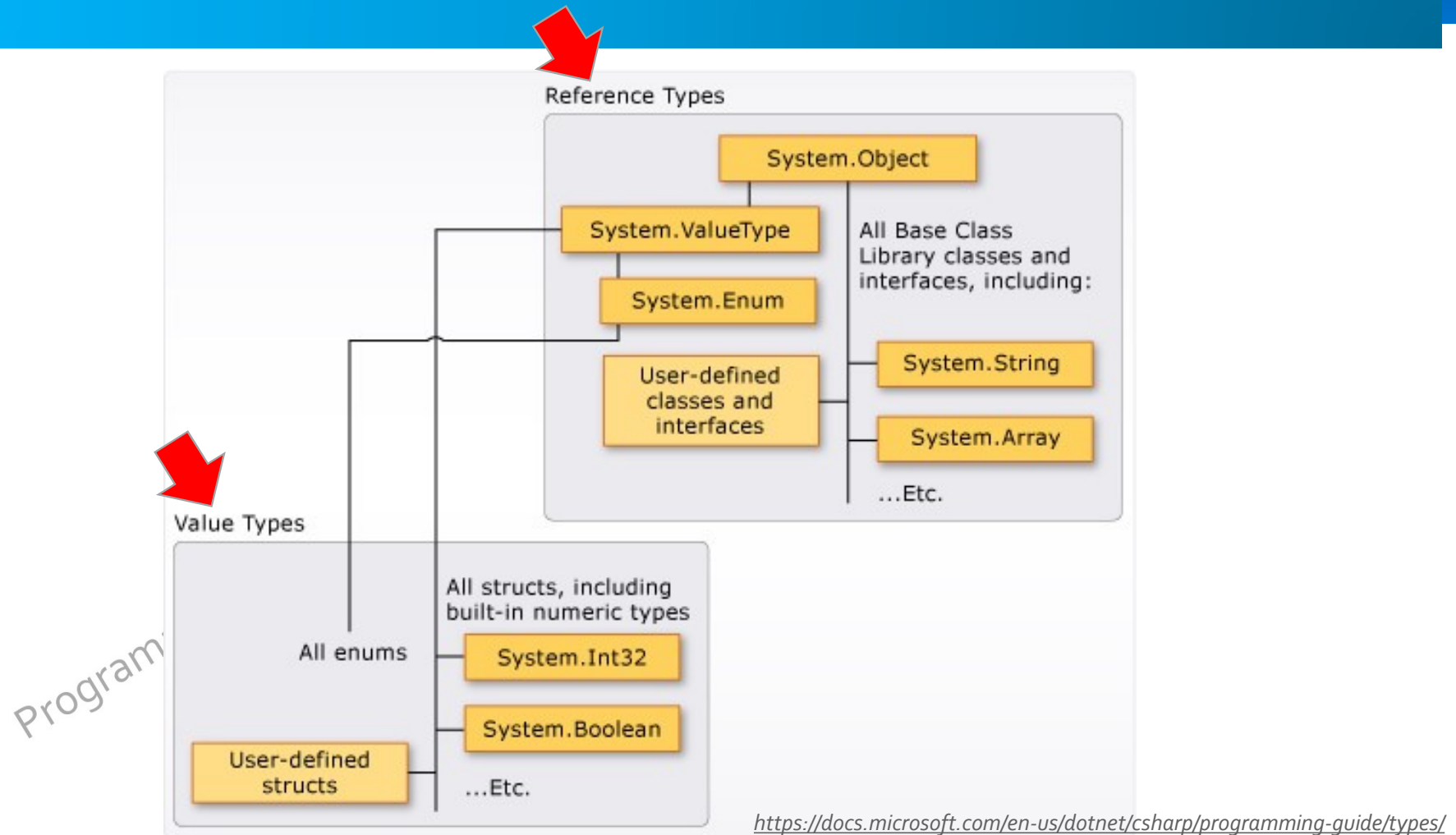
- Especificados com a palavra chave *dynamic*
- Podem conter qualquer valor (*string*, *número*, *objeto*, *referência* a uma função/método)
- As operações são avaliadas em *runtime*

```
dynamic val1 = 1;  
dynamic val2 = 2;  
Console.WriteLine(val1 + val2); // Escreve 3  
val1 = "1";  
val2 = 2;  
Console.WriteLine(val1 + val2); // Escreve 12
```

Tipos de Dados em C#

- Tipos de dados em C# são categorizados pela forma como são armazenados em memória. Existem dois tipos:
 - Valor
 - *bool, byte, char, decimal, double, enum, float, int, long, sbyte, short, struct, uint, ulong, ushort*
 - Referência
 - *String, Arrays, Classes e Delegates* (Veremos mais adiante)

Tipos de Dados em C#



Programação Web

C#: Enums, Structs, Arrays

Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra



C#: Enumerations

- Tipo de variável que permite declarar uma lista de constantes, podendo assim ser utilizados de uma forma *human-readable*
- Muitas vezes referidos como **enums**
- Sintaxe:

```
enum <nome>
{
    lista
};
```

 - *nome*: especifica o nome do tipo de enumeração
 - *lista*: É uma lista de identificadores separados por vírgula

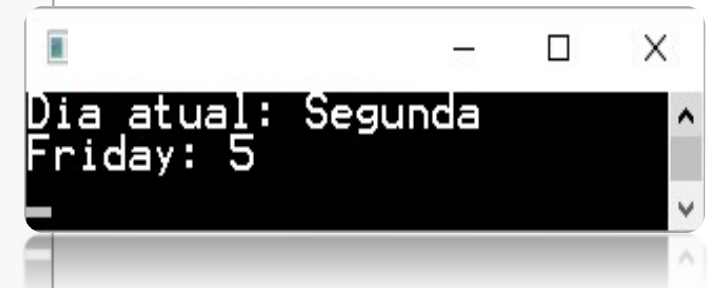
C#: Enumerations - Exemplo

```
enum DiasSemana : int { Segunda=1,
                        Terca=2,
                        Quarta=3,
                        Quinta=4,
                        Sexta=5,
                        Sabado=7,
                        Domingo=8
                        };

static void Main(string[] args)
{
    DiasSemana diaSemanaAtual = DiasSemana.Segunda;
    int FimSemana = (int)DiasSemana.Sexta;
    Console.WriteLine("Dia atual: {0}", diaSemanaAtual);
    Console.WriteLine("Friday: {0}", FimSemana);
    Console.ReadKey();
}
```

@JB

Saída?



C#: Enumerations - Exemplo

```
using System;
public class Enum
{
    enum DiasSemana { Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado};

    static void Main()
    {
        int x = (int)DiasSemana.Domingo;
        int y = (int)DiasSemana.Sexta;
        Console.WriteLine("Domingo = {0}", x);
        Console.WriteLine("Sexta = {0}", y);
        Console.ReadKey();
    }
}
```

Execute no VS e veja o resultado!

C#: Struct

- **Class** é tipo *referência* \neq **Struct** é tipo *valor*
- Estruturas podem conter construtores, constantes, métodos, propriedades, eventos,...
- Usa-se a **keyword** **struct**
- Pode ser inicializada com ou sem o **new**
- Estruturas não podem incluir constructores ou destrutores *default*, mas podem ter construtores parametrizados

C#: Struct

- Podem implementar interfaces
- Não podem herdar outra estrutura ou classe
 - Membros das estruturas não podem ser especificados como ***abstract***, ***virtual***, ou ***protected***.
 - Devem ser inicializadas com o ***new*** de forma a poder usar-se as suas propriedades, métodos ou eventos
- Um ***struct*** pode ser usado como um tipo que permite valor nulo e receber um valor nulo.

C#: Struct - Exemplo

```
using System;

class Program
{
    static void Main(string[] args)
    {
        Carro carro;

        carro = new Carro("Azul");
        Console.WriteLine(carro.Descricao());

        carro = new Carro("Vermelho");
        Console.WriteLine(carro.Descricao());

        Console.ReadKey();
    }
}
```

```
struct Carro {
    private string cor;

    public Carro(string cor)
    {
        this.cor = cor;
    }

    public string Descricao()
    {
        return "Cor do carro = " + cor;
    }

    public string Cor
    {
        get { return cor; }
        set { cor = value; }
    }
}
```

C#: Arrays

Numero[0]	Numero[1]	Numero[2]	Numero[3]	...
-----------	-----------	-----------	-----------	-----

```
int[] tabela;  
int[] numeros;  
numeros = new int[10];  
numeros = new int[20];  
string[,] saladeaula;  
saladeaula = new string[10, 6];  
int[,,] cubo;
```


C#: Arrays

```
int[] numeros= new int[5] { 1, 2, 3, 4, 5 };  
string[] nomes= new string[3] { "Joana", "Filipa", "José" };
```

// ou

```
int[] numeros = new int[] { 1, 2, 3, 4, 5 };  
string[] nomes = new string[] { "Joana", "Filipa", "José" };
```

// ou

```
int[] numeros = { 1, 2, 3, 4, 5 };  
string[] nomes = { "Joana", "Filipa", "José" };
```

C#: Arrays – Acesso aos Elementos

```
int[] n = new int[10];
int i, j;

for (i = 0; i < 10; i++)
{
    n[i] = i + 100;
}

for (j = 0; j < 10; j++)
{
    Console.WriteLine("Elemento[{0}] = {1}", j, n[j]);
}
```

C#: Arrays – Acesso aos Elementos

▪ *foreach*

```
int[] n = new int[10];

for (int i = 0; i < 10; i++)
{
    n[i] = i + 100;
}

foreach (int j in n)
{
    int i = j - 100;
    Console.WriteLine("Elemento[{0}] = {1}", i, j);
}

Console.ReadKey();
```

Programação Web

C#: Namespaces

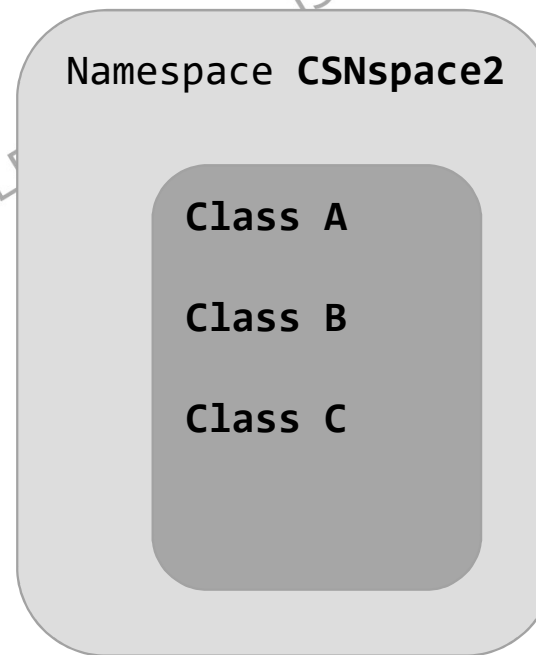
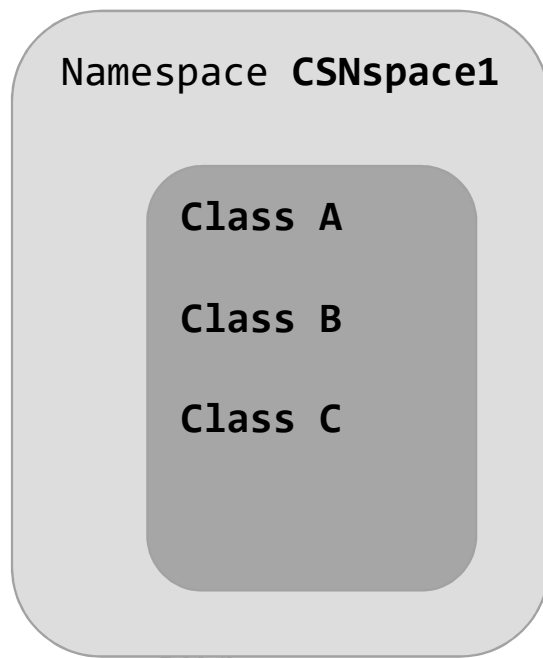
Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra



C#: Namespaces

- O Código dos programas C# estão organizados em ***namespaces***
 - Organiza um conjunto de classes relacionadas entre si
 - As classes podem ter nomes iguais, desde que estejam em *namespaces* diferentes
 - Um *namespace* pode conter outros *namespaces*

C#: Namespaces



CSNspace1.A ...

CSNspace2.A ...

C#: Namespaces

```
using System;  
...  
Console.Write("Introduza um nome:");  
string nome = Console.ReadLine();
```

```
System.Console.Write("Introduza um numero:");  
string nome = System.Console.ReadLine();
```

C#: Namespaces

```
namespace Pweb.Exemplo
{
    class ClasseExemplo
    {
    }
}
```

Programação Web

Programação Orientada Objetos com C#

*Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra*



Programação Orientada a Objetos

- Representação de cada elemento em termos de um objeto, ou classe.
- Aproximar o sistema que se cria com o que se observa no mundo real
- Reutilização de código
 - Temporal e número de linhas de código
 - Independência
- Facilidade de leitura e manutenção de código, ...

POO – Princípios Básicos

Abstração

Encapsulamento

***Programação
Orientada a
Objetos***

Herança

Polimorfismo

Programação Orientada a Objectos

- Auxilia na solução de problemas aproximando o sistema que se cria com o que se observa do mundo real;
- Dada a complexidade do mundo real, é necessário abstrair conhecimento relevante e encapsular dentro de **objetos**;
- Conjunto de objetos trabalham juntos para realizar uma tarefa;

Programação Orientada a Objectos

- O mundo é composto por diversos **objetos** que possuem um conjunto de **características** e um **comportamento** bem definido;
- Definir **abstrações** dos objetos reais existentes;
- Todo **objeto** possui as seguintes características:
 - *Estado*: conjunto de propriedades de um objeto;
 - *Comportamento*: conjunto de ações possíveis sobre o objeto;
 - *Unicidade*: todo objeto é único.

Programação Orientada a Objectos

- A comunicação entre objectos é efetuada por mensagens

- **Objecto = Dados + Código**

Dados → atributos
Código → métodos

- **Classes**

- Definem os tipos de objectos
- Definem quais os dados e o código dos objectos

Programação Orientada a Objectos

Classe



Objectos

As **classes** são a caracterização abstrata de algo (cliente, empregado, carro,...)

Objeto \neq Classe

Objeto = Instância de uma Classe

Programação Orientada a Objectos

```
public class Carro
{
    private string cor;
    private string modelo;
    private string anoFabrico;
    private string combustivel;
    public void Anda()
    {
        ...
    }
    public void Para()
    {
        ...
    }
    public void Acelera()
    {
        ...
    }
}
```



C#: Objetos

- Para criação de um objeto:
 - **Definição da Classe**
 - Considerada como um *template* do objeto
 - Exemplo: *Template* de um *Carro*
 - **Criação do objeto**
 - Objetos de uma classe criados em *runtime*
 - Exemplo: *Carro* em si! Instância do Carro

C#: Classes

- Sintaxe geral:

```
acesso class nome_da_classe
{
    // Bloco de Instruções
}
```

- **acesso**

- nível de proteção da classe (*internal* / *public*)
- Por omissão, assume o nível *internal*
- Pode ser também *abstract* ou *sealed*

C#: Classes

```
using System;
namespace Exemplo {
    class Program {
        static void Main(string[] args)
        {
            @override x = new @override();
            Console.WriteLine(x.Numero);
            Console.ReadKey();
        }
    }
    class @override
    {
        public int Numero { get; set; }

        public @override()
        {
            Numero = 5;
        }
    }
}
```

Programação

C#: Classes

- *Construtor*
 - Se não for definido um construtor para uma classe, o C# define um por omissão

```
class MinhaClasse {  
    public MinhaClasse()  
    {  
        // Construtor Default  
    }  
  
    public MinhaClasse(int meuInt)  
    {  
        // Construtor com parâmetro  
    }  
}
```

C#: Classes

- *Construtor: keyword **this***

```
class Motociclo {  
    public string nomeCondutor;  
    public int velocidade;  
    public Motociclo() {}  
    public Motociclo(int velocidade): this(velocidade, "") { }  
    public Motociclo(string nome) : this(0,nome) { }  
    public Motociclo(int velocidade, string nome) {  
        if(velocidade>20) { velocidade = 20; }  
        this.velocidade = velocidade;  
        nomeCondutor = nome;  
    }  
}
```

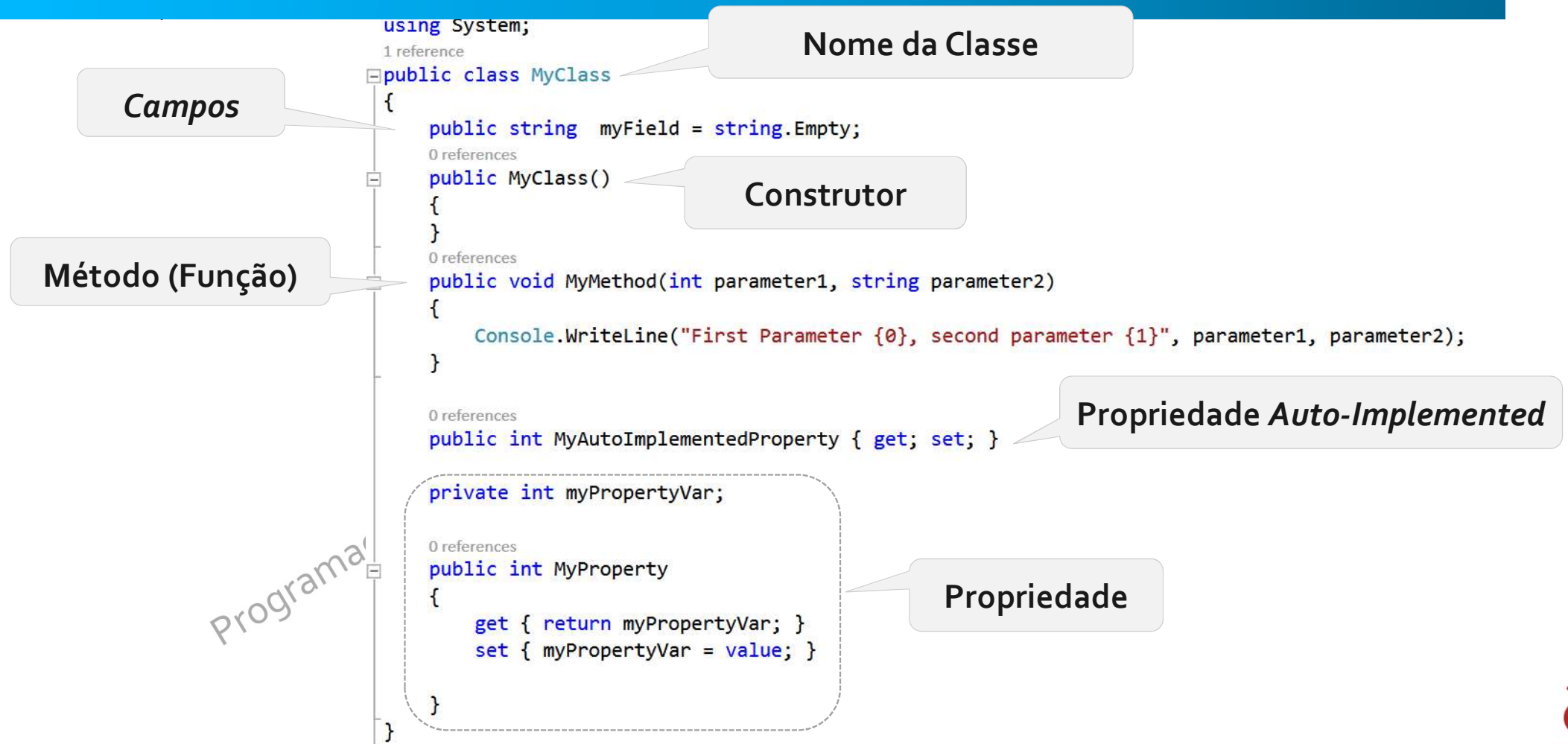
C#: Classes

- *Destrutor*

```
class MinhaClasse
{
    ~MinhaClasse()
    {
        //Destrutor
    }
}
```

- Executado quando o *Garbage Collector* ocorre, permitindo libertar recursos

C#: Classe



C#: Membros de uma Classe

- Definindo uma classe, fornece-se definições para todos os membros:
 - Variáveis onde são armazenados os dados, designados como **campos** ou **atributos** (*fields*);
 - **Métodos** que permitem manipular os dados;
 - **Propriedades**
 - Recorrendo aos *getters* e *setters*

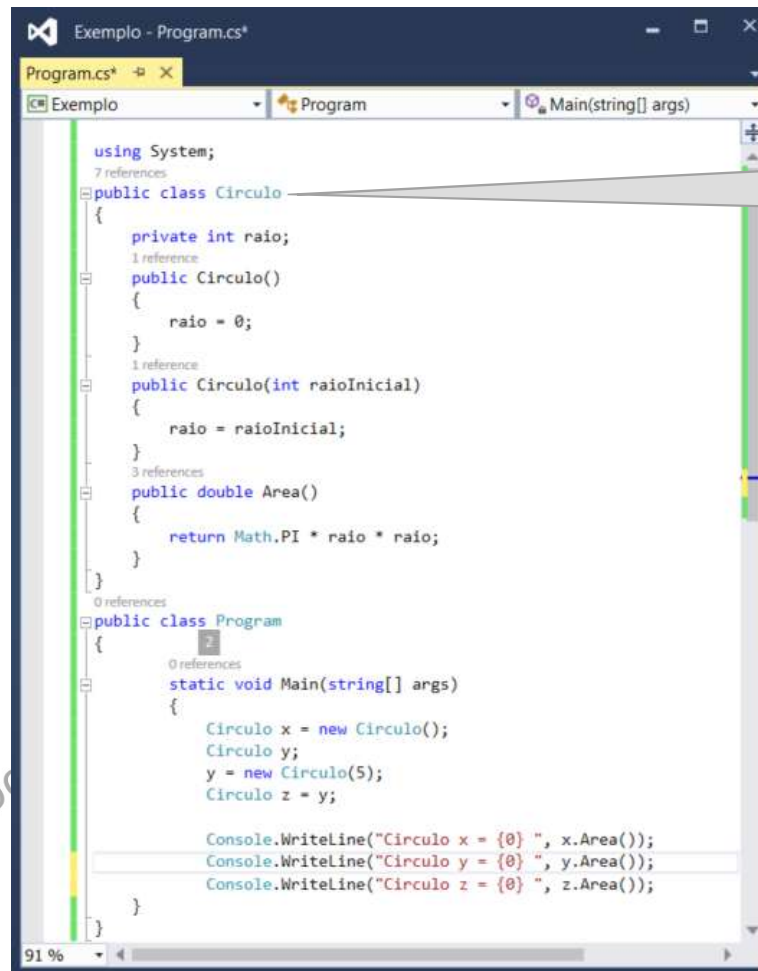
C#: Atributos / Campos

- *Atributos* são também designados como *campos* (*member fields*)
- Permitem definir as características/dados que especificam os objetos
- Em C# os atributos de uma classe são definidos no interior da mesma
- Sintaxe: **acesso** tipo *field*;
 - **acesso**: Nível de proteção do atributo
private (por omissão), *public*, *protected*, *internal*, *protected internal*

C#: Campos

Nível de proteção	Descrição
private (por omissão)	Só pode ser acedido a partir da classe onde foi declarado
public	Pode ser acedido externamente por outras classes ou métodos (incluindo fora do assembly onde foi definido)
protected	Só pode ser acedido a partir da classe onde foi declarado e a partir de classes derivadas
internal	Pode ser acedido a partir de qualquer classe (exceto fora do assembly onde foi definido)
protected internal	Só pode ser acedido a partir da classe onde foi declarado ou partir de classes derivadas (exceto fora do assembly onde foi definido)

C#: Exemplo (1)



```
using System;

public class Circulo
{
    private int raio;

    public Circulo()
    {
        raio = 0;
    }

    public Circulo(int raioInicial)
    {
        raio = raioInicial;
    }

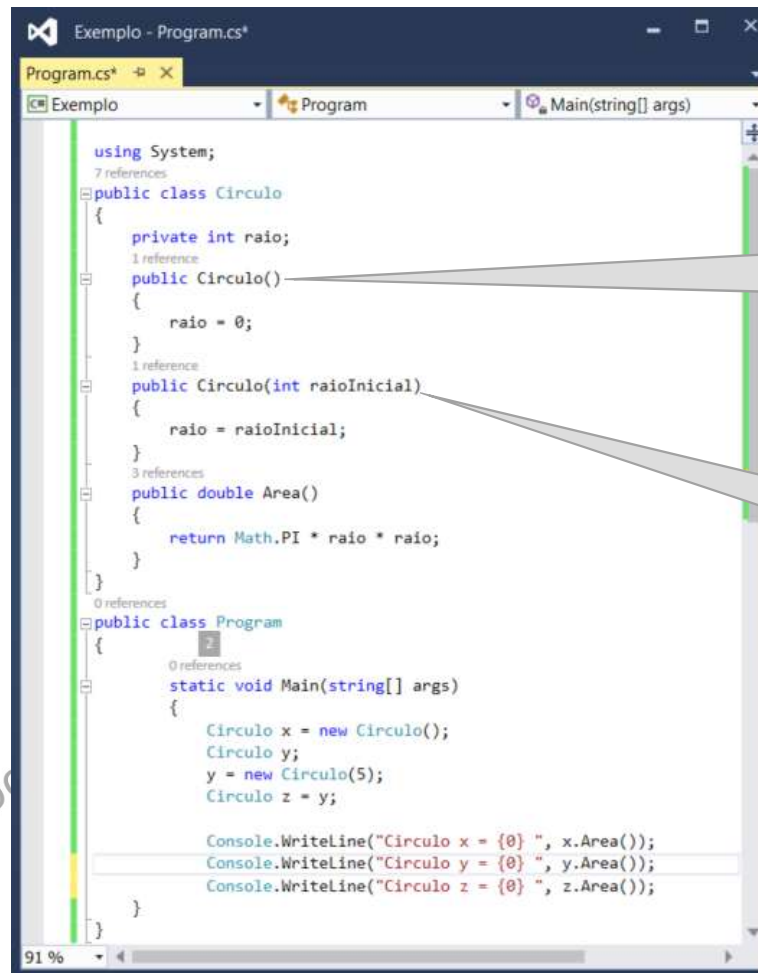
    public double Area()
    {
        return Math.PI * raio * raio;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Circulo x = new Circulo();
        Circulo y;
        y = new Circulo(5);
        Circulo z = y;

        Console.WriteLine("Circulo x = {0} ", x.Area());
        Console.WriteLine("Circulo y = {0} ", y.Area());
        Console.WriteLine("Circulo z = {0} ", z.Area());
    }
}
```

Criação da classe **Circúlo** com o campo *raio*

C#: Exemplo (1)



```
using System;

public class Circulo
{
    private int raio;

    public Circulo()
    {
        raio = 0;
    }

    public Circulo(int raioInicial)
    {
        raio = raioInicial;
    }

    public double Area()
    {
        return Math.PI * raio * raio;
    }
}

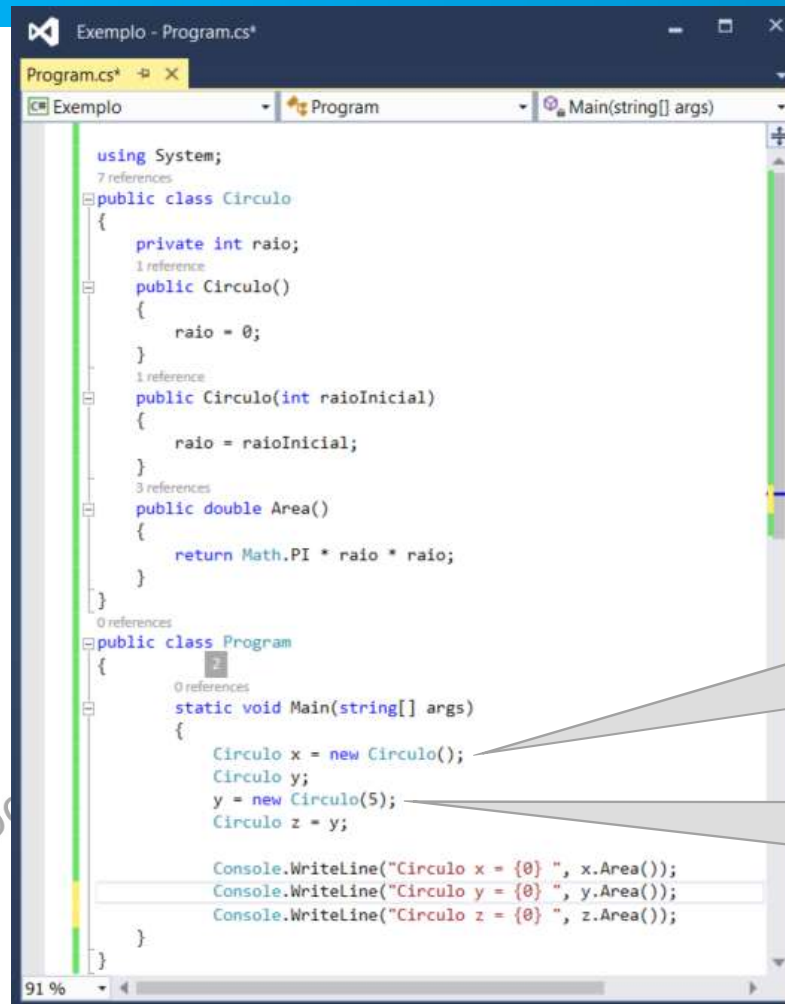
public class Program
{
    static void Main(string[] args)
    {
        Circulo x = new Circulo();
        Circulo y;
        y = new Circulo(5);
        Circulo z = y;

        Console.WriteLine("Circulo x = {0} ", x.Area());
        Console.WriteLine("Circulo y = {0} ", y.Area());
        Console.WriteLine("Circulo z = {0} ", z.Area());
    }
}
```

Construtor *default* da classe **Circulo**

Construtor da classe **Circulo** passando por parâmetro o raio

C#: Exemplo (1)



```
using System;

public class Circulo
{
    private int raio;

    public Circulo()
    {
        raio = 0;
    }

    public Circulo(int raioInicial)
    {
        raio = raioInicial;
    }

    public double Area()
    {
        return Math.PI * raio * raio;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Circulo x = new Circulo();
        Circulo y;
        y = new Circulo(5);
        Circulo z = y;

        Console.WriteLine("Circulo x = {0} ", x.Area());
        Console.WriteLine("Circulo y = {0} ", y.Area());
        Console.WriteLine("Circulo z = {0} ", z.Area());
    }
}
```

Criação de uma **instância** da classe **Circulo** (uso da keyword **new**)

Criação de uma **instância** da classe **Circulo**

C#: Exemplo (1)

```
using System;
7 references
public class Circulo
{
    private int raio;
    1 reference
    public Circulo()
    {
        raio = 0;
    }
    1 reference
    public Circulo(int raioInicial)
    {
        raio = raioInicial;
    }
    3 references
    public double Area()
    {
        return Math.PI * raio * raio;
    }
}
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        Circulo x = new Circulo();
        Circulo y;
        y = new Circulo(5);
        Circulo z = y;

        Console.WriteLine("Circulo x = {0} ", x.Area());
        Console.WriteLine("Circulo y = {0} ", y.Area());
        Console.WriteLine("Circulo z = {0} ", z.Area());
    }
}
```

Saída?

```
Circulo x = 0
Circulo y = 78,5398163397448
Circulo z = 78,5398163397448
```

C#: *Partial Class*

- Pode-se “partir” a classe em vários ficheiros
 - O nome dos ficheiros pode ser qualquer um, importante é definir o tipo (neste caso classe) com o mesmo nome
- Recurso à keyword **partial**
- *Visual Studio* recorre constantemente a classes parciais
- Por exemplo:
 - Métodos e Propriedades num ficheiro e restantes elementos noutro...

C#: Partial Class

```
namespace PartialClasse
{
    partial class MinhaClasse
    {
        // Campos e Construtores
        private int idade;
        public MinhaClasse()
        {
            idade = 0;
        }
        public MinhaClasse(int i)
        {
            idade = i;
        }
    }
}
```

MinhaClasse.cs

```
namespace PartialClasse
{
    partial class MinhaClasse
    {
        // Metodos e Propriedades
        public int Idade
        {
            get { return idade; }
            set {
                if (value < 0 || value > 100)
                    idade = 0;
                else idade = value;
            }
        }
    }
}
```

MinhaClasse.Core.cs

C#: Níveis de Proteção das Classes

Nível de Proteção	Descrição
internal (por omissão)	Classe acessível apenas no projeto corrente
public	Classe acessível de qualquer parte
abstract internal abstract	Acessível apenas no projeto corrente e não pode ser instanciada, apenas derivada de
public abstract	Pode ser acessível de qualquer parte e não pode ser instanciada, apenas derivada de
sealed internal sealed	Acessível apenas no projeto corrente e não pode ser derivada de, apenas instanciada
public sealed	Acessível de qualquer parte e não pode ser derivada de, apenas instanciada

C#: Encapsulamento

- **Encapsulamento** tem como objetivo a ocultação de pormenores internos da classe
- Fornecida uma *interface pública* constituída por métodos que são a única coisa que o resto do programa se apercebe da classe
- A implementação da classe pode ser modificada sem que isso interfira com o resto do programa, desde que não se altere a interface

POO: Encapsulamento

- Adicionam segurança na implementação em OO uma vez que escondem elementos internos, criando uma espécie de *black box*
- Maior parte das linguagens OO implementam o encapsulamento com recurso a métodos especiais chamados de *getters* e *setters*
- Em C# as **propriedades** evitam o acesso direto aos campos do objeto

C#: Propriedades

- Definidas de forma semelhante aos campos, permitindo desempenhar um processamento adicional
 - Recurso às *keywords* **get** e **set**
- Estrutura básica:

```
public int MinhaPropriedade
{
    get {
        // código para o get
    }
    set {
        // código para o set
    }
}
```

C#: Propriedades

- Propriedade
 - Encapsula um campo privado
 - Permite especificar o *get* para retribuir valores de um campo e permite especificar o *set* para definir um valor
 - *Exemplo:*

```
private int meuCampo;  
public int MeuCampo  
{  
    get { return meuCampo;  
    }  
    set { meuCampo=value;  
    }  
}
```

C#: Propriedades

- É possível também adicionar lógica nas propriedades:

```
class Exemplo
{
    private int propriedade;
    public int Propriedade {
        get
        {
            return propriedade / 2;
        }
        set
        {
            if (value > 100) propriedade = 100;
            else propriedade = value;
        }
    }
}
```

Saída?

```
Exemplo x = new Exemplo();
c.Propriedade = 1;
```

C#: Propriedades

```
using System;
class IntervaloTempo
{
    private double segundos;

    public double Horas
    {
        get { return segundos / 3600; }
        set {
            if (value < 0 || value > 24)
                throw new ArgumentOutOfRangeException(
                    $"{nameof(value)} deve estar compreendido entre 0 e 24.");
            segundos = value * 3600;
        }
    }
}
```

IntervaloTempo t = new IntervaloTempo();
t.Horas = 24;
Console.WriteLine(\$"Tempo em horas: {t.Horas}");

C#: *Auto Implemented Properties*

- *Auto implemented Properties* em C#
 - Surgiram com o C# 3.0
 - Permitem uma declaração simplificada, quando não é necessária especificar qualquer lógica no *get* ou *set*
 - Não foram declaradas variáveis internas?
 - Criadas pelo compilador! Poupa tempo e torna o código mais limpo.

```
public String MeuNome  
{  
    get; set;  
}
```


C#: Auto Implemented Properties

```
using System;
public class Aluno
{
    public string Nome { get; set; }
    public decimal Idade { get; set; }
}
class Program
{
    static void Main(string[] args)
    {
        var aluno = new Aluno{ Nome = "xpto", Idade = 17 };
        Console.WriteLine($"{aluno.Nome} tem {aluno.Idade} anos!");
    }
}
```

C#: Propriedades

- Apenas de leitura

```
public int Numero {get;}
```

- Apenas de escrita

```
public int Numero {set;}
```



C#: Properties

```
public class Aluno
{
    string nome;
    decimal idade;
    public Aluno(string nome, decimal idade)
    {
        this.nome = nome;
        this.idade = idade;
    }
    public string Nome
    {
        get => nome;
        set => nome = value;
    }
    public decimal Idade
    {
        get => idade;
        set => idade = value;
    }
}
```

```
var aluno = new Aluno{ Nome = "xpto", Idade = 17 };
Console.WriteLine($"{aluno.Nome} tem {aluno.Idade} anos!");
```

C# 7.0

Expression-bodied members (EBM's)

O C# 6 permitiu a implementação de propriedades somente leitura e métodos através das expressões lambdas e isso foi chamado de EBM.

Na versão 7.0, este recurso foi melhorado permitindo a utilização de expressões lambdas em construtores, propriedades com get/set e finalizadores.

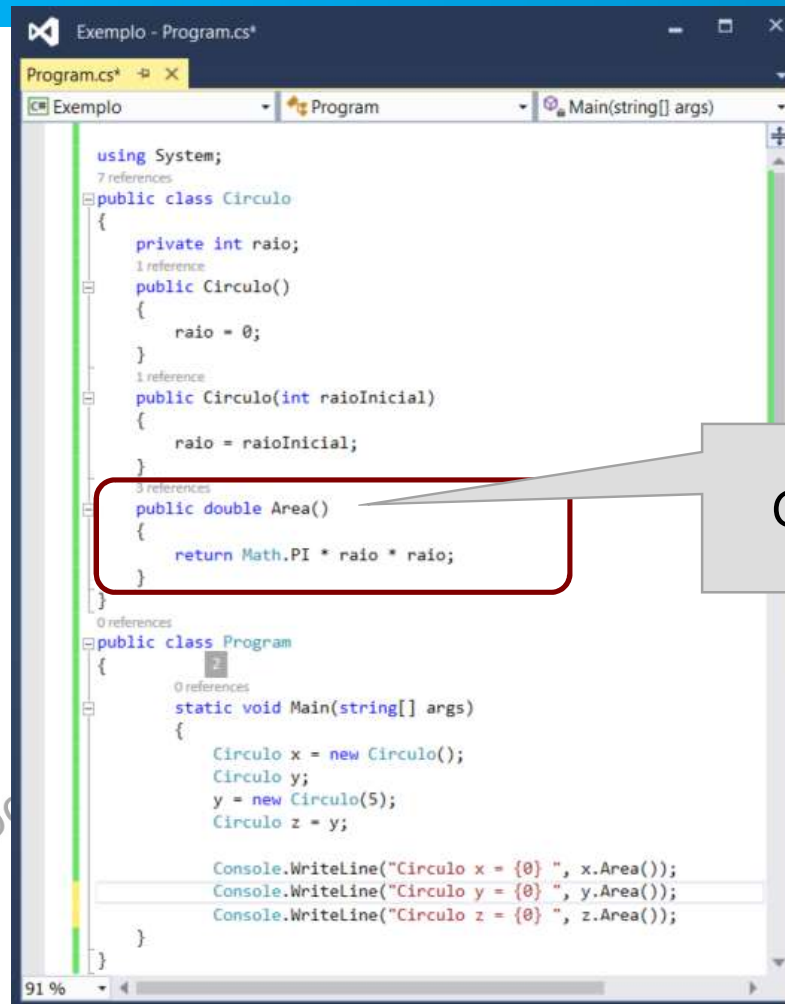
O objetivo do recurso é tornar o código conciso e legível fazendo com que o membro do tipo (construtor, métodos, propriedades, destrutor, etc) seja definido em uma única expressão.

C#: Métodos

- Ações que os objetos podem desempenhar, definidos dentro do corpo de uma classe
- Tecnicamente conhecidas por funções de membro de uma classe
 - Sintaxe: `acesso tipo nome_método(argumentos)`
 - Exemplo:

```
public void fazQualquerCoisa() { }  
  
public String ObtemNome() { }
```
- Existem métodos próprios: construtores, ...

C#: Exemplo



```
using System;

public class Circulo
{
    private int raio;

    public Circulo()
    {
        raio = 0;
    }

    public Circulo(int raioInicial)
    {
        raio = raioInicial;
    }

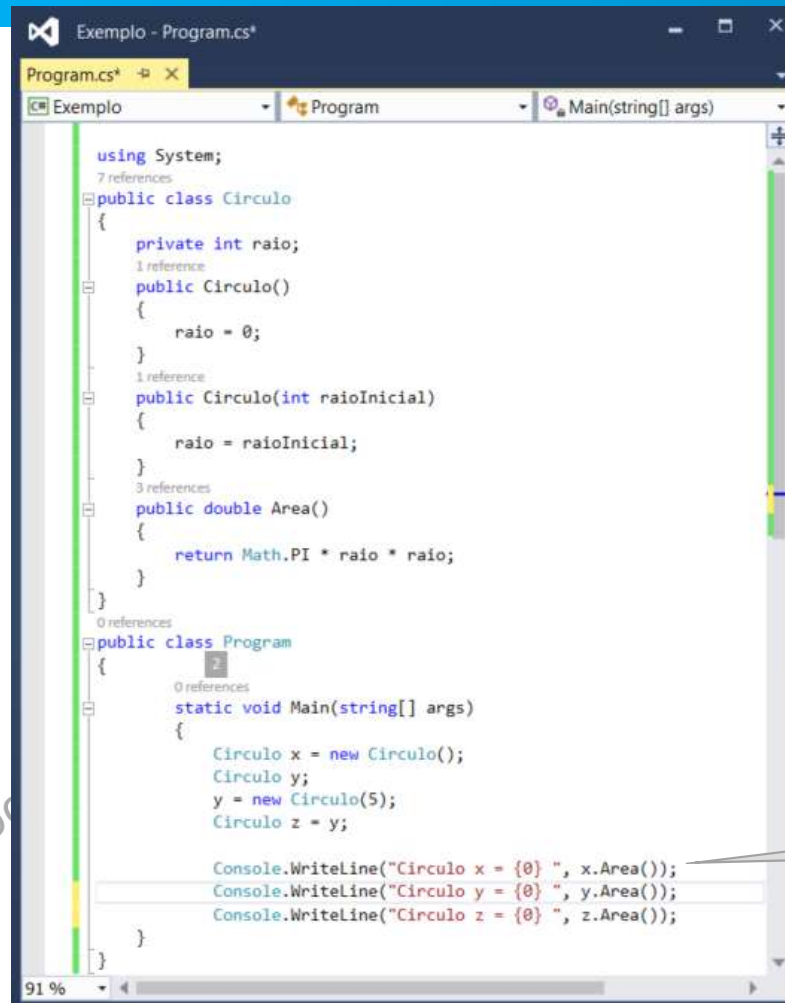
    public double Area()
    {
        return Math.PI * raio * raio;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Circulo x = new Circulo();
        Circulo y;
        y = new Circulo(5);
        Circulo z = y;

        Console.WriteLine("Circulo x = {0} ", x.Area());
        Console.WriteLine("Circulo y = {0} ", y.Area());
        Console.WriteLine("Circulo z = {0} ", z.Area());
    }
}
```

Criação do método **Area()**

C#: Exemplo



```
using System;

public class Circulo
{
    private int raio;

    public Circulo()
    {
        raio = 0;
    }

    public Circulo(int raioInicial)
    {
        raio = raioInicial;
    }

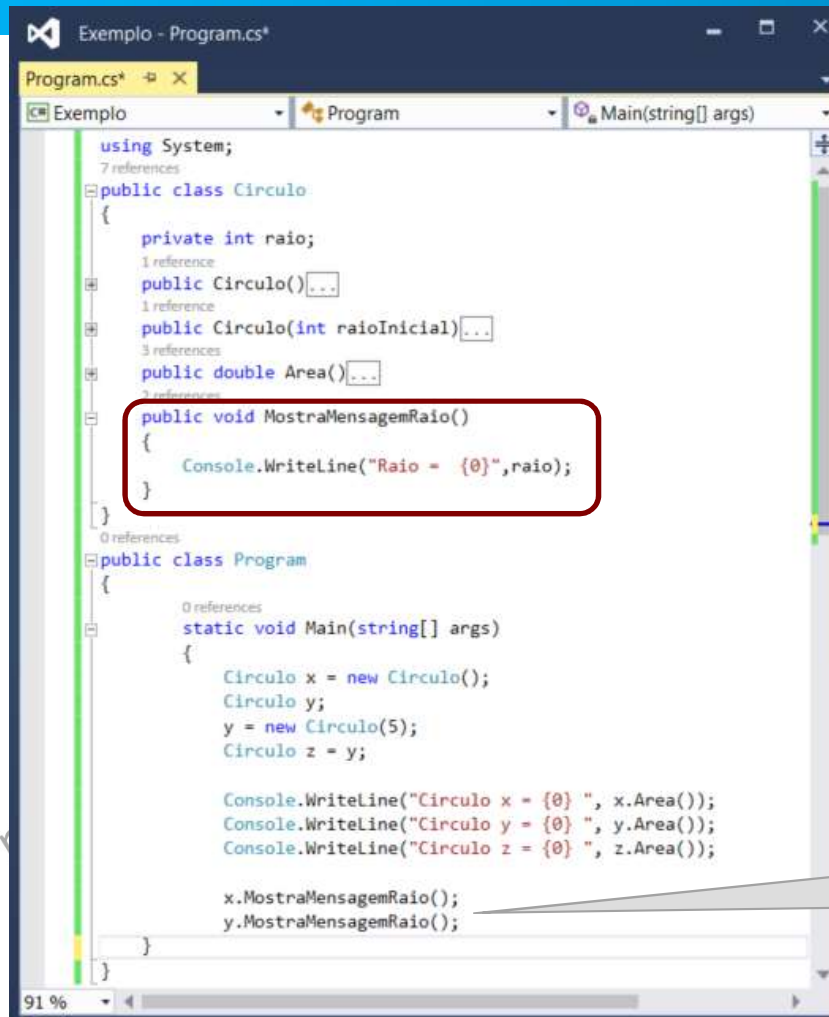
    public double Area()
    {
        return Math.PI * raio * raio;
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Circulo x = new Circulo();
        Circulo y;
        y = new Circulo(5);
        Circulo z = y;

        Console.WriteLine("Circulo x = {0} ", x.Area());
        Console.WriteLine("Circulo y = {0} ", y.Area());
        Console.WriteLine("Circulo z = {0} ", z.Area());
    }
}
```

Chamada ao método **Area**

C#: Exemplo



```
using System;

public class Circulo
{
    private int raio;

    public Circulo()...
    public Circulo(int raioInicial)...
    public double Area()...
    public void MostraMensagemRaio()
    {
        Console.WriteLine("Raio = {0}", raio);
    }
}

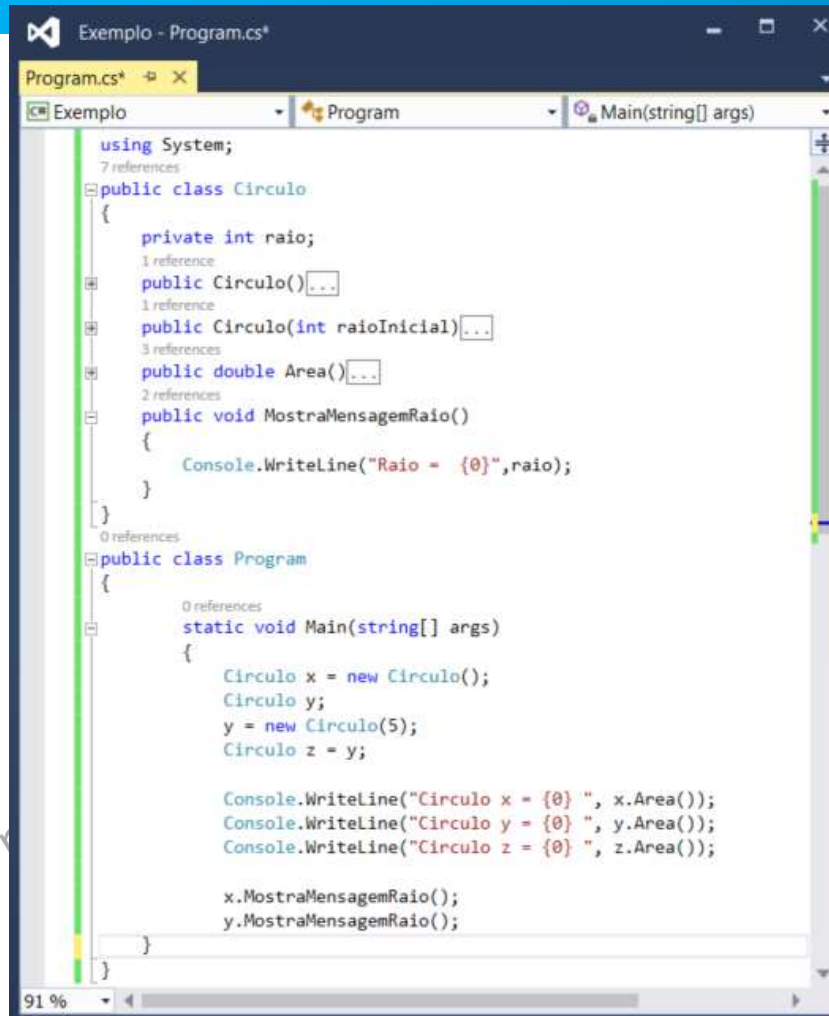
public class Program
{
    static void Main(string[] args)
    {
        Circulo x = new Circulo();
        Circulo y;
        y = new Circulo(5);
        Circulo z = y;

        Console.WriteLine("Circulo x = {0} ", x.Area());
        Console.WriteLine("Circulo y = {0} ", y.Area());
        Console.WriteLine("Circulo z = {0} ", z.Area());

        x.MostraMensagemRaio();
        y.MostraMensagemRaio();
    }
}
```

Chamada do método
MostraMensagemRaio

C#: Exemplo



```
using System;

public class Circulo
{
    private int raio;

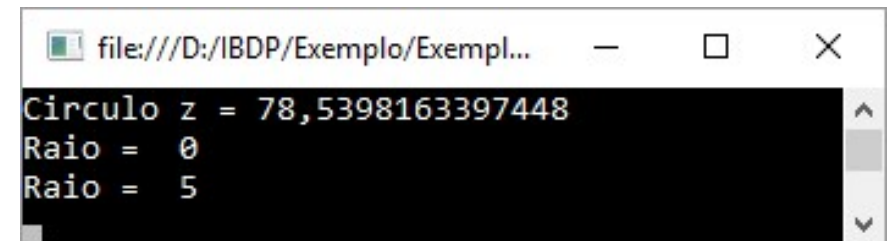
    public Circulo()...
    public Circulo(int raioInicial)...
    public double Area()...
    public void MostraMensagemRaio()
    {
        Console.WriteLine("Raio = {0}", raio);
    }
}

public class Program
{
    static void Main(string[] args)
    {
        Circulo x = new Circulo();
        Circulo y;
        y = new Circulo(5);
        Circulo z = y;

        Console.WriteLine("Circulo x = {0} ", x.Area());
        Console.WriteLine("Circulo y = {0} ", y.Area());
        Console.WriteLine("Circulo z = {0} ", z.Area());

        x.MostraMensagemRaio();
        y.MostraMensagemRaio();
    }
}
```

Saída?



```
file:///D:/IBDP/Exemplo/Exempl...
Circulo z = 78,5398163397448
Raio = 0
Raio = 5
```


C#: Métodos - Parâmetros

- Por valor
- Por referência
 - *ref*

```
using System;

class PassagemParametros
{
    1 reference
    static void Quadrado(int x)
    {
        x *= x;
        System.Console.WriteLine("Valor dentro do metodo: {0}", x);
    }
    0 references
    static void Main(string[] args)
    {
        int n = 5;
        Console.WriteLine("Valor antes de chamar o método: {0}", n);

        Quadrado(n);
        Console.WriteLine("Valor depois de chamar o método: {0}", n);

        Console.ReadKey();
    }
}
```

Método com parâmetros

C#: Passagem de Parâmetros por valor

- Por valor
- Por referência
 - *ref*

```
using System;
0 references
class PassagemParametros
{
    1 reference
    static void Quadrado(int x)
    {
        x *= x;
        System.Console.WriteLine("Valor dentro do metodo: {0}", x);
    }
    0 references
    static void Main(string[] args)
    {
        int n = 5;
        Console.WriteLine("Valor antes de chamar o método: {0}", n);

        Quadrado(n);
        Console.WriteLine("Valor depois de chamar o método: {0}", n);

        Console.ReadKey();
    }
}
```

C#: Passagem de Parâmetros por valor

- Por valor
- Por referência
 - *ref*

```
using System;
0 references
class PassagemParametros
{
    1 reference
    static void Quadrado(int x)
    {
        x *= x;
        System.Console.WriteLine("Valor dentro do metodo: {0}", x);
    }
    0 references
    static void Main(string[] args)
    {
        int n = 5;
        Console.WriteLine("Valor antes de chamar o método: {0}", n);

        Quadrado(n);
        Console.WriteLine("Valor depois de chamar o método: {0}", n);

        Console.ReadKey();
    }
}
```

Valor antes de chamar o método: 5
Valor dentro do metodo: 25
Valor depois de chamar o método: 5

Programação Web

C#: Passagem de Parâmetros *ref*

- Valores
- Referência
 - *ref*
 - Devem ser inicializados antes de serem passados ao método

```
using System;
0 references
class PassagemParametros
{
    1 reference
    static void Quadrado(ref int x)
    {
        x *= x;
        System.Console.WriteLine("Valor dentro do metodo: {0}", x);
    }
    static void Main(string[] args)
    {
        int n = 5;
        Console.WriteLine("Valor antes de chamar o método: {0}", n);

        Quadrado(ref n);
        Console.WriteLine("Valor depois de chamar o método: {0}", n);

        Console.ReadKey();
    }
}
```

Programação Web

C#: Passagem de Parâmetros por referência

- Valores
- Referência
 - *ref*

Saída?

```
Valor antes de chamar o método: 5
Valor dentro do método: 25
Valor depois de chamar o método: 25
```

```
using System;
0 references
class PassagemParametros
{
    1 reference
    static void Quadrado(ref int x)
    {
        x *= x;
        System.Console.WriteLine("Valor dentro do método: {0}", x);
    }
    0 references
    static void Main(string[] args)
    {
        int n = 5;
        Console.WriteLine("Valor antes de chamar o método: {0}", n);

        Quadrado(ref n);
        Console.WriteLine("Valor depois de chamar o método: {0}", n);
        Console.ReadKey();
    }
}
```

C#: Métodos – Parâmetros *out*

- **out**
- Não necessita ser inicializado antes de ser passado ao método

```
using System;  
0 references
```

Programação Web

```
0 references  
static void Main(string[] args)  
{  
    int[] meuArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };  
    int maxIndice;  
    Console.WriteLine($"O valor máximo no meuArray é { MaxValor(meuArray, out maxIndice)}");  
    Console.WriteLine($"A primeira ocorrencia deste valor está no elemento { maxIndice + 1}");  
  
    Console.ReadKey();  
}
```


C#: Métodos – Parâmetros *out*

- *out*

```
using System;
0 references
class Parametros
{
    1 reference
    static int MaxValor(int[] intArray, out int maxIndice)
    {
        int maxVal = intArray[0];
        maxIndice = 0;
        for (int i = 1; i < intArray.Length; i++)
        {
            if (intArray[i] > maxVal)
            {
                maxVal = intArray[i];
                maxIndice = i;
            }
        }
        return maxVal;
    }
    0 references
    static void Main(string[] args)
    {
        int[] meuArray = { 1, 8, 3, 6, 2, 5, 9, 3, 0, 2 };
        int maxIndice;
        Console.WriteLine($"O valor máximo no meuArray é { MaxValor(meuArray, out maxIndice)}");
        Console.WriteLine($"A primeira ocorrência deste valor está no elemento { maxIndice + 1}");
    }
}
```

O valor máximo no meuArray é 9
A primeira ocorrência deste valor está no elemento 7

Programação Web

C#: Parâmetros *ref* vs *out*

```
using System;
namespace RefVSOut {
    class Program
    {
        static void Main(string[] args)
        {
            int val1 = 0;
            int val2;

            MethodRef(ref val1);
            Console.WriteLine(val1);

            MethodOut(out val2);
            Console.WriteLine(val2);
        }
    }
    ...
}
```

```
static void MethodRef(ref int value)
{
    value = 1;
}
static void MethodOut(out int value)
{
    value = 2;
}
```



C#: *Params Arrays*

- C# suporta ***parameter arrays***
- Permite a passagem de um número variável de parâmetros de tipo idêntico (ou classes relacionadas por herança) como um único parâmetro lógico
- Argumentos especificados com a keyword *params* podem ser processados, se chamado através do envio de um *array* fortemente “tipado” ou uma lista de itens separados por vírgula

C#: *Params Arrays*

```
using System;
namespace ArrayApplication {
    class ParamArray {
        public int AddElements(params int[] arr) {
            int sum = 0;
            foreach (int i in arr)
                sum += i;
            return sum;
        }
    }
    class TestClass {
        static void Main(string[] args) {
            ParamArray app = new ParamArray();
            int sum = app.AddElements(512, 720, 250, 567, 889);
            Console.WriteLine("Soma: {0}", sum);
            Console.ReadKey();
        }
    }
}
```

C#: Params Arrays

```
using System;
namespace ArrayApplication {
    class ParamArray {
        public int AddElements(params int[] arr) {
            int sum = 0;
            foreach (int i in arr)
                sum += i;
            return sum;
        }
    }
    class TestClass {
        static void Main(string[] args) {
            ParamArray app = new ParamArray();
            int sum = app.AddElements();
            Console.WriteLine("Soma: {0}", sum);
            Console.ReadKey();
        }
    }
}
```

C#: Params Arrays

```
using System;
namespace ArrayApplication {
    class ParamArray {
        public int AddElements(params int[] arr) {
            int sum = 0;
            foreach (int i in arr)
                sum += i;
            return sum;
        }
    }
    class TestClass {
        static void Main(string[] args) {
            ParamArray app = new ParamArray();
            int sum = app.AddElements(int[] dados={1, 4, 5, 6 });
            Console.WriteLine(AddElements(dados));
            Console.ReadKey();
        }
    }
}
```

C#: Parâmetros Opcionais

- O C# permite criar métodos que passam **argumentos opcionais**
- Permite invocação do método omitindo alguns argumentos “desnecessários”

Programação Web – 2023/2024 – FUD+CEPL – ISEC/IPC @FEUP

C#: Parâmetros Opcionais - Exemplo

```
class Program {  
    static void Main() {  
        // Omite os parâmetros opcionais  
        Metodo();  
        // Omite segundo parametro  
        Metodo(4);  
        // Sintaxe clássica  
        Metodo(4, "Jose");  
        // ERRO: Metodo("Filomena"); Necessário especificar o nome do parâmetro  
        Metodo(nome: "Filipa");  
        // Especifica ambos os parâmetros  
        Metodo(nome: "Maria", numero: 5);  
    }  
    static void Metodo(int numero = 0, string nome = "Indefinido") {  
        System.Console.WriteLine("Numero = {0}, Nome = {1}", numero, nome);  
    }  
}
```

C#: Parâmetros Opcionais



CE+PL – ISEC/IPC @JB

```
static void Metodo(int dia = System.DateTime.Now.Day ,
                    string nome = "Indefinido")
{
    System.Console.WriteLine("Dia = {0}, Nome = {1}",
                             dia, nome);
}
```

Programação Web

C#: Polimorfismo, Herança, ...

Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra



C#: Polimorfismo

- O **Polimorfismo** permite ao programador definir **métodos genéricos** em classes ou interfaces, que podem ser implementados de **diversas formas** nas respectivas subclasses
- Polimorfismo aplica-se apenas aos métodos da classe e, por definição, exige a utilização de herança
- Vantagens:
 - Reutilização de código;
 - Tempo de desenvolvimento;
 - Manutenção.

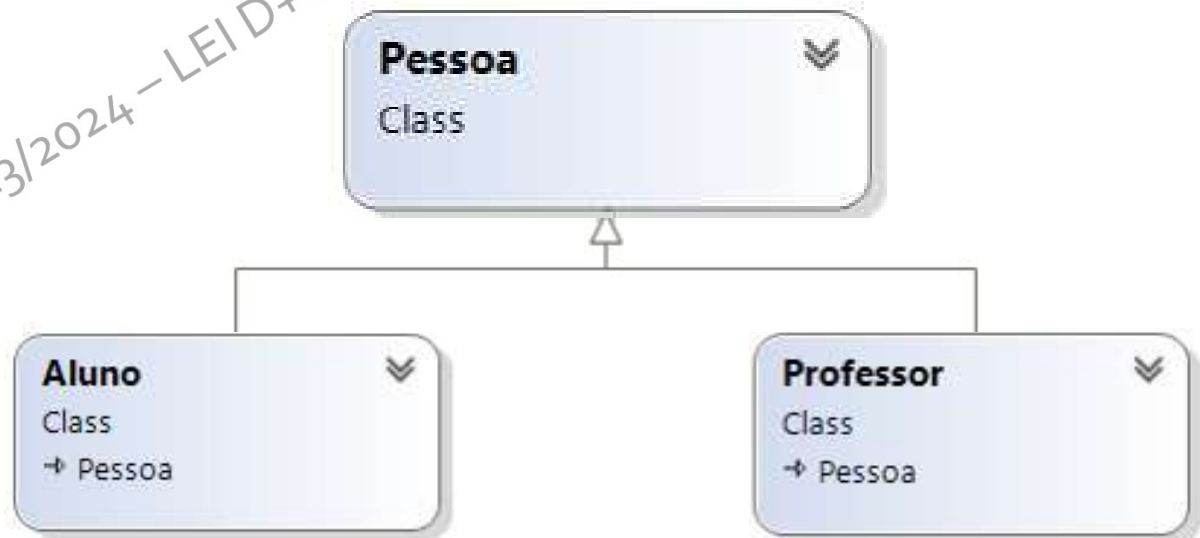
C#: Herança

- A **Herança** é um dos principais conceitos da POO
- Permite reutilização de código e redução da complexidade
- Permite **construção de classes baseadas noutras classes** já existentes (neste caso, a primeira classe diz-se *classe base*)
- A subclasse:
 - Herda todas as propriedades e métodos da classe existente
 - Pode incluir ou sobrepor novas propriedades e novos métodos

C#: Herança

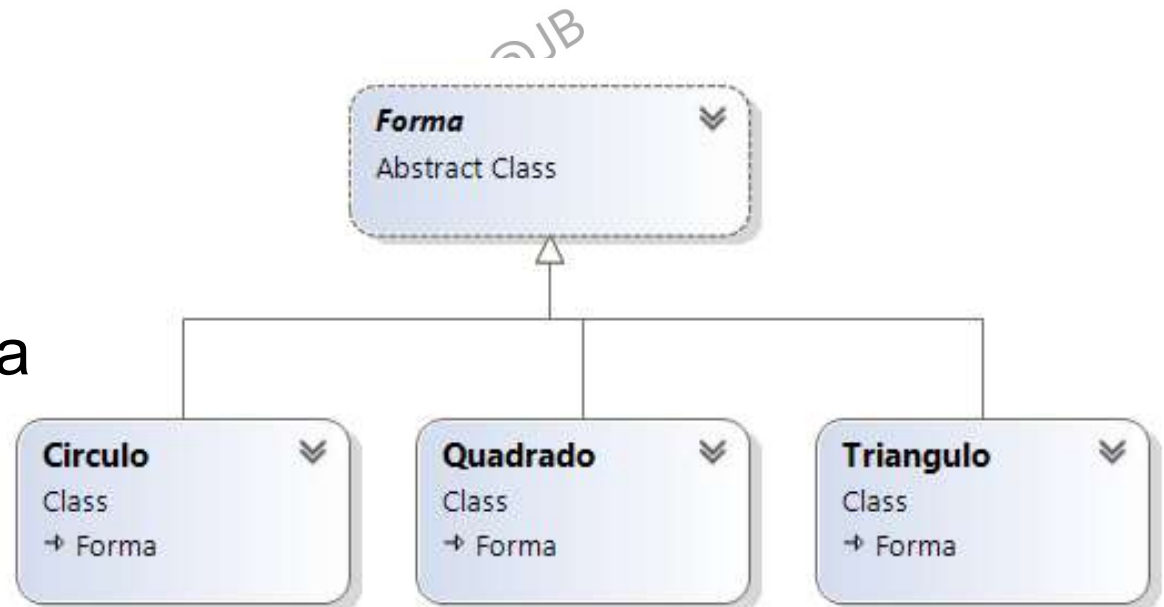
- Em C# apenas é permitido **uma classe base!**
- Mecanismo recursivo, permitindo criar uma hierarquia de classes

- Exemplo:

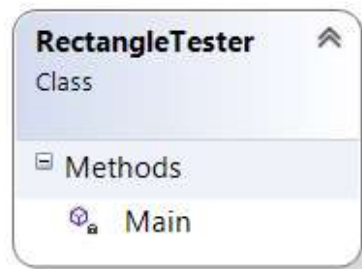
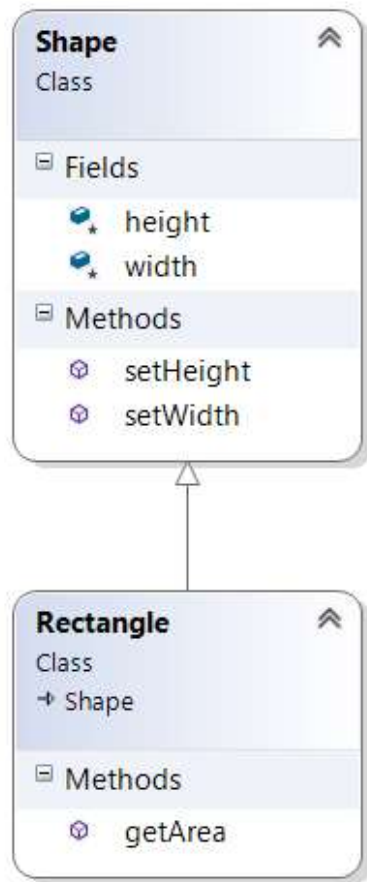


C#: Herança (3)

- Exemplo 2:
 - Um “quadrado” é uma “forma”
 - Uma “forma” define uma propriedade comum “cor”
 - Um “quadrado” herda a propriedade “cor”

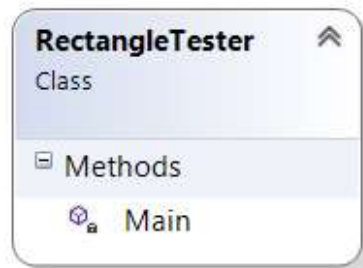
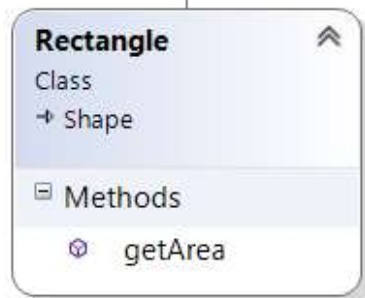
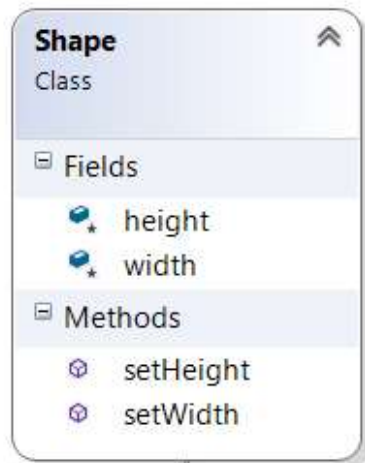


C#: Herança – Exemplo (1)



```
class Shape
{
    public void setWidth(int w)
    {
        width = w;
    }
    public void setHeight(int h)
    {
        height = h;
    }
    protected int width;
    protected int height;
}
```

C#: Herança – Exemplo (1)



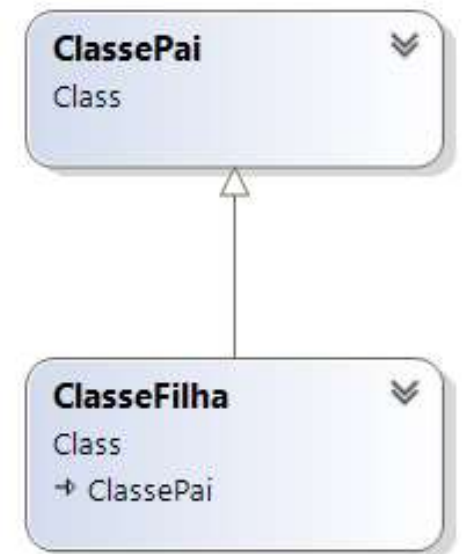
```
class Rectangle : Shape
{
    public int getArea()
    {
        return (width * height);
    }
}
```

```
class RectangleTester
{
    static void Main(string[] args)
    {
        Rectangle Rect = new Rectangle();
        Rect.setWidth(5); Rect.setHeight(7);
        Console.WriteLine("Total area: {0}", Rect.getArea());
    }
}
```

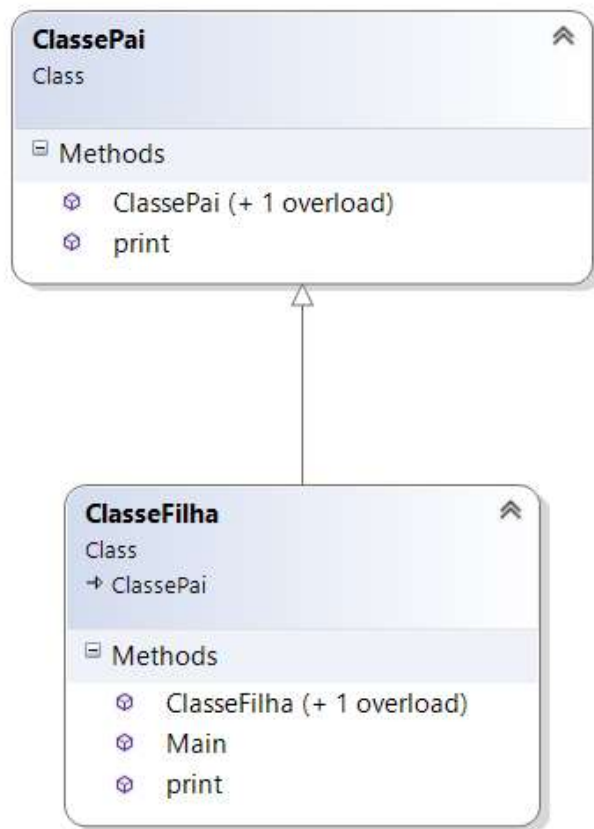
C#: Herança

- Herdar uma classe

```
public class ClassePai
{
    // Membros da classe
}
internal class ClasseFilha: ClassePai
{
    // Membros da classe
}
```



C#: Herança



```
public class ClassePai
{
    public ClassePai()
    {
        Console.WriteLine("Construtor Pai");
    }
    public ClassePai(string texto)
    {
        Console.WriteLine("Construtor Pai " + texto);
    }
    public void print()
    {
        Console.WriteLine("Print() da classe Pai.");
    }
}
```


C#: Herança

Ocultar qualquer implementação do método print() acima dele

```
public class ClasseFilha : ClassePai    {
    public ClasseFilha() : base("Mensagem Ola...") {
        Console.WriteLine("Construtor Filha");
    }
    public ClasseFilha(string x) : base(x) {
        Console.WriteLine("Construtor Filha com string");
    }
    public new void print() {
        base.print();
        Console.WriteLine("Print() da classe filha");
    }
    public static void Main() {
        ClasseFilha filha = new ClasseFilha();
        ClasseFilha filha2 = new ClasseFilha("String Parametro");
        filha.print();
        ((ClassePai)filha).print();
    }
}
```

C#: Herança

- Membros da classe **base** com acesso **private** não são acessíveis da classe derivada
 - Nível de proteção **protected** permite que as classes derivadas tenham acesso ao membro, sem o tornar público
- Membros da classe base podem ser virtuais (**virtual**)
 - A classe que herda uma classe com membros virtuais, pode ser **overridden** pela classe que a herda -> i.e. altera o método original existente na classe pai
 - Isso, permite que a classe derivada possa ter uma implementação alternativa para o membro.

C#: Classes e Métodos Abstratos

- A Classe *Base* pode ser **Abstrata**
 - Uma Classe Abstrata é uma classe conceptual no qual se define funcionalidades para que as subclasses (que a herdam) possam implementá-las de forma **não obrigatória**
- Uma *classe abstrata* pode ser somente herdada e não instanciada
- Classes abstratas podem ter membros abstratos, os quais não tem implementação na classe base, portanto, tem de ser fornecida uma implementação na classe derivada caso sejam usados na classe derivada

C#: Classes e Métodos Abstratos(2)

- Quando uma classe possui pelo menos um método abstrato esta deve também ser declarada como abstrata
- Classes bases abstratas podem já fornecer implementação dos membros
- Uma classe abstrata pode herdar de outra classe abstrata

```
public abstract class nome_da_classe
{
    // Membros da classe, podem ser abstratos
}
```

C#: Classes e Métodos Abstratos(3)

- Um método abstrato é um método que ***não possui implementação*** na classe abstrata, apenas possui a definição da sua **assinatura**
- A sua implementação deve ser feita na **classe derivada**
- Os métodos podem ou não ser abstratos, mas quando estes são abstratos, a sua **implementação é obrigatória!**

C#: Membros virtuais

- Membro ***virtual***
 - É um membro na classe base que define uma **implementação por omissão** que pode ser alterada (*overridden*) pela classe derivada
 - **Por contraste, um método abstrato**, é um membro na classe base que não fornece qualquer implementação por omissão, mas fornece a sua assinatura.

Classes Abstratas e *Sealed*

- A palavra chave **abstract** permite criar classes e membros de classes incompletas que têm de ser implementados na classe derivada
- A palavra chave **sealed** evita que a classe ou membros virtuais sejam herdados

C#: Interfaces

- Para a definição de **Interfaces**, utiliza-se a palavra chave **interface**

```
public interface IMeuInterface
{
    // Membros da Interface
}
```

- Todas as interfaces são **públicas**
- Na versão do C# 9, os métodos declarados numa interface podem possuir implementação e são sempre públicos
- Quando uma interface é referenciada tem de se implementar a funcionalidade dos seus métodos

C#: Interfaces (1)

```
public class MyClass : IUmaInterface
{
    // Membros da class
}
```

```
public class MyClass : MyBase, IUma1Interface, IUma2Interface
{
    // Membros da class
}
```

C#: Herança e Interfaces

- Herdar uma *Interface*

```
public class MyClass : IMeuInterface
{
    // Membros da class
}
```

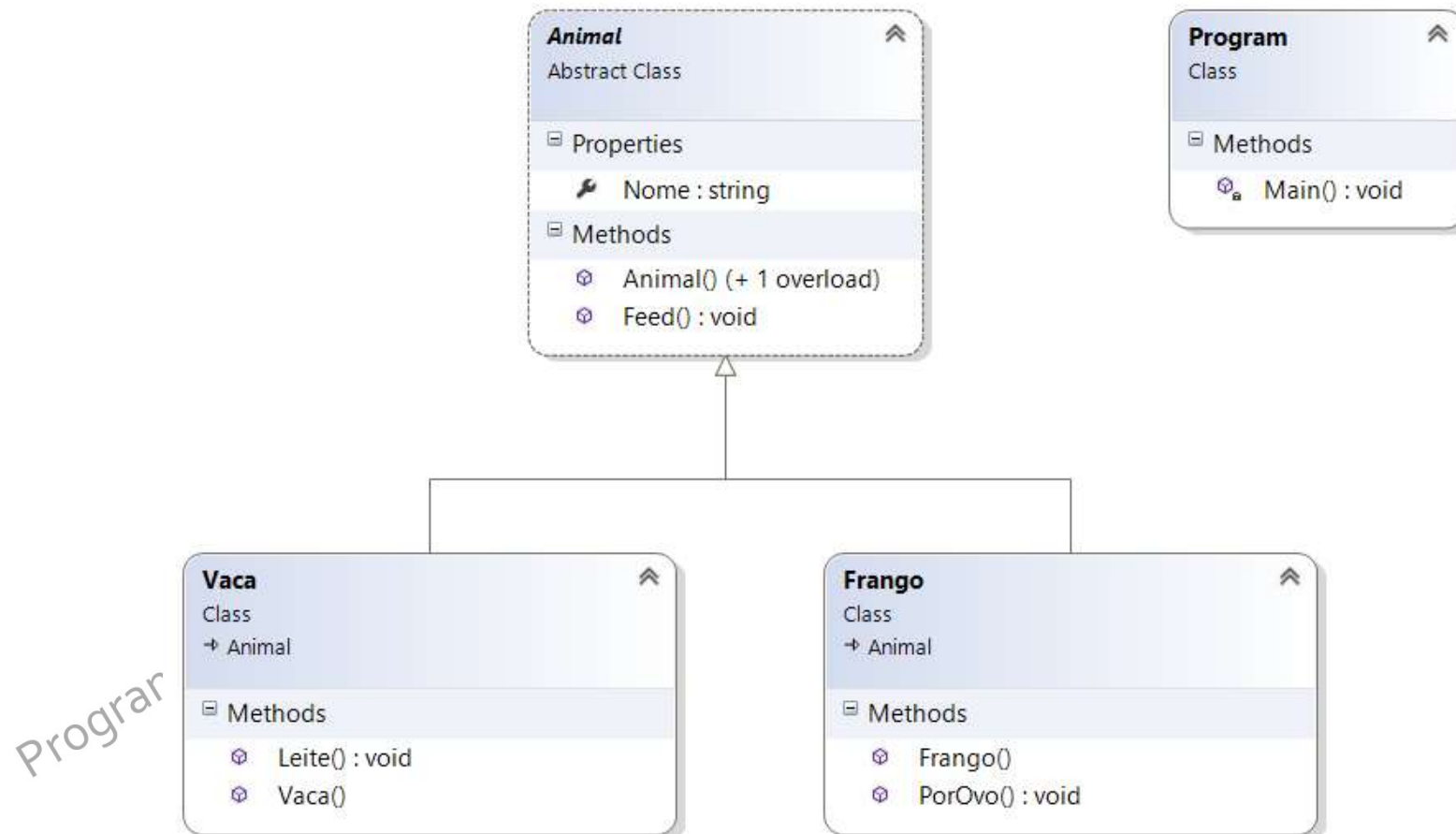
```
public class MyClass : MyBase, IMeuInterface, IMeu2Interface
{
    // Membros da class
}
```

C#: Interfaces

Implicitamente
public e abstract

```
public interface IMeuInterface  
{  
    // Membros da Interface  
    int método();  
}
```

C#: Classes e Métodos Abstratos - Exemplo



C#: Classes e Métodos Abstratos - Exemplo

```
using System;
namespace CSharpExemplo
{
    public abstract class Animal
    {
        public string Nome { get; set; }
        public Animal()
        {
            Nome = "Animal sem nome";
        }
        public Animal(string novoNome)
        {
            Nome = novoNome;
        }
        public void Feed()
        {
            Console.WriteLine($"{Nome} foi alimentado!");
        }
    }
}
```

C#: Classes e Métodos Abstratos - Exemplo

```
using System;
namespace CSharpExemplo
{
    public abstract class Animal
    {
        public string Nome { get; set; }
        public Animal()
        {
            Nome = "Animal sem nome";
        }
        public Animal(string novoNome)
        {
            Nome = novoNome;
        }
        public void Feed()
        {
            Console.WriteLine($"{Nome} foi alimentado!");
        }
    }
}
```

```
using System;
namespace CSharpExemplo {
    public class Frango : Animal {
        public void PorOvo() {
            Console.WriteLine($"{nome} pôs um ovo.");
        }
        public Frango(string novoNome) : base(novoNome) { }
    }
}
```

```
using System;
namespace CSharpExemplo {
    public class Vaca : Animal
    {
        public void Leite() {
            Console.WriteLine($"A vaca {nome} foi ordenhada! ");
        }
        public Vaca(string novoNome) : base(novoNome) {}
    }
}
```

C#: Classes e Métodos Abstratos - Exemplo

```
static void Main(string[] args)
{
    Vaca minhaVaca = new Vaca("LeaVaca");
    Frango meuFrango2 = new Frango("ZetaFrango");
    Animal meuFrango = new Frango("NoaFrango");
    Console.WriteLine($"Criei 3 objectos: {minhaVaca.Nome},
                        {meuFrango.Nome} e {meuFrango2.Nome}");

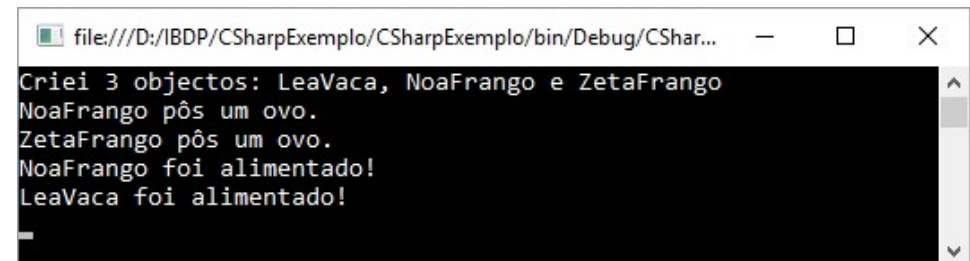
    meuFrango2.PorOvo();
    ((Frango)meuFrango).PorOvo();
    meuFrango.Feed();
    minhaVaca.Feed();
}
```

C#: Classes e Métodos Abstratos - Exemplo

```
static void Main(string[] args)
{
    Vaca minhaVaca = new Vaca("LeaVaca");
    Frango meuFrango2 = new Frango("ZetaFrango");
    Animal meuFrango = new Frango("NoaFrango");
    Console.WriteLine($"Criei 3 objectos: {minhaVaca.Nome},
                        {meuFrango.Nome} e {meuFrango2.Nome}");

    meuFrango2.PorOvo();
    ((Frango)meuFrango).PorOvo();
    meuFrango.Feed();
    minhaVaca.Feed();
}
```

Saída?



```
file:///D:/IBDP/CSharpExemplo/CSharpExemplo/bin/Debug/CShar...
Criei 3 objectos: LeaVaca, NoaFrango e ZetaFrango
NoaFrango pôs um ovo.
ZetaFrango pôs um ovo.
NoaFrango foi alimentado!
LeaVaca foi alimentado!
```

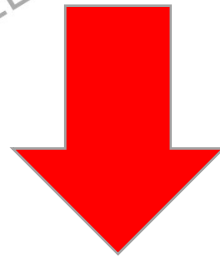

C#: Classes e Métodos Abstratos - Exemplo

```
namespace CSharpExemplo {  
    public abstract class Animal  
    {  
        protected string nome;  
        public string Nome  
        {  
            get { return nome; }  
            set { nome = value; }  
        }  
        public Animal()  
        {  
            nome = "Animal sem nome";  
        }  
        public Animal(string novoNome)  
        {  
            nome = novoNome;  
        }  
        public virtual void Feed();  
    }  
}
```



C#: Classes e Métodos Abstratos - Exemplo

```
namespace CSharpExemplo {  
    public abstract class Animal  
    {  
        protected string nome;  
        public string Nome  
        {  
            get { return nome; }  
            set { nome = value; }  
        }  
        public Animal()  
        {  
            nome = "Animal sem nome";  
        }  
        public Animal(string novoNome)  
        {  
            nome = novoNome;  
        }  
  
        public virtual void Feed() {  
            Console.WriteLine($"{Nome} foi alimentado!");  
        }  
    }  
}
```



C#: Classes e Métodos Abstratos - Exemplo

```
using System;
namespace CSharpExemplo {
    public class Frango : Animal
    {
        public void PorOvo() {
            Console.WriteLine($"{Nome} pôs um ovo.");
        }
        public Frango(string novoNome) : base(novoNome){ }

        public override void Feed()
        {
            Console.WriteLine($"{Nome} foi alimentado com milho!");
        }
    }
}
```

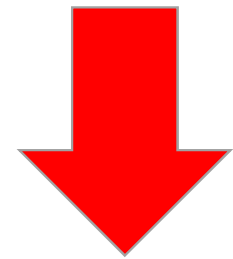
C#: Classes e Métodos Abstratos - Exemplo

```
using System;
namespace CSharpExemplo {
    public class Animal {
        {
            public virtual void Feed() {
                Console.WriteLine($"{Nome} pôs um ovo.");
            }
        }
        public virtual void Feed() {
            Console.WriteLine($"{Nome} pôs um ovo.");
        }
        public virtual void Feed() {
            Console.WriteLine($"{Nome} pôs um ovo.");
        }
    }
}

public override void Feed()
{
    Console.WriteLine($"{Nome} foi alimentado com milho!");
}
```

C#: Classes e Métodos Abstratos - Exemplo

```
namespace CSharpExemplo {  
    public abstract class Animal  
    {  
        protected string nome;  
        public string Nome  
        {  
            get { return nome; }  
            set { nome = value; }  
        }  
        public Animal()  
        {  
            nome = "Animal sem nome";  
        }  
        public Animal(string novoNome)  
        {  
            nome = novoNome;  
        }  
        public abstract void Feed();  
    }  
}
```



É OBRIGATÓRIO!

C#: Classes e Métodos Abstratos - Exemplo

```
using System;
namespace CSharpExemplo {
    public class Frango : Animal
    {
        public void PorOvo() {
            Console.WriteLine($"{Nome} pôs um ovo.");
        }
        public Frango(string novoNome) : base(novoNome){ }

        public override void Feed()
        {
            Console.WriteLine($"{Nome} foi alimentado com milho!");
        }
    }
}
```

C#: Classes e Métodos Abstratos - Exemplo

```
using System;
namespace CSharpExemplo {
    public class Vaca : Animal
    {
        public void Leite() {
            Console.WriteLine($"A vaca {nome} foi ordenhada! ");
        }
        public Vaca(string novoNome) : base(novoNome) { }

        public override void Feed()
        {
            Console.WriteLine($"{Nome} foi alimentado com erva!");
        }
    }
}
```

C#: Classes e Métodos Abstratos - Exemplo

```
static void Main(string[] args)
{
    Vaca minhaVaca = new Vaca("LeaVaca");
    Frango meuFrango = new Frango("ZetaFrango");
    Animal meuAnimal = new Animal("NovoAnimal");
    object meuObjecto = new Frango("FrangoObjecto");
    Animal meuFrango2 = new Frango("NoaFrango");
    meuFrango.PorOvo();
    meuFrango2.PorOvo();
}
```



C#: Classes e Métodos Abstratos - Exemplo

```
static void Main(string[] args)
{
    Vaca minhaVaca = new Vaca("LeaVaca");
    Frango meuFrango = new Frango("ZetaFrango");
    Animal meuAnimal = new Animal("NovoAnimal");
    object meuObjecto = new Frango("FrangoObjecto");
    Animal meuFrango2 = new Frango("NoaFrango");
    meuFrango.PorOvo();
    meuFrango2.PorOvo(); ((Frango)meuFrango2).PorOvo();
}
```



C#: Será que isto compila?

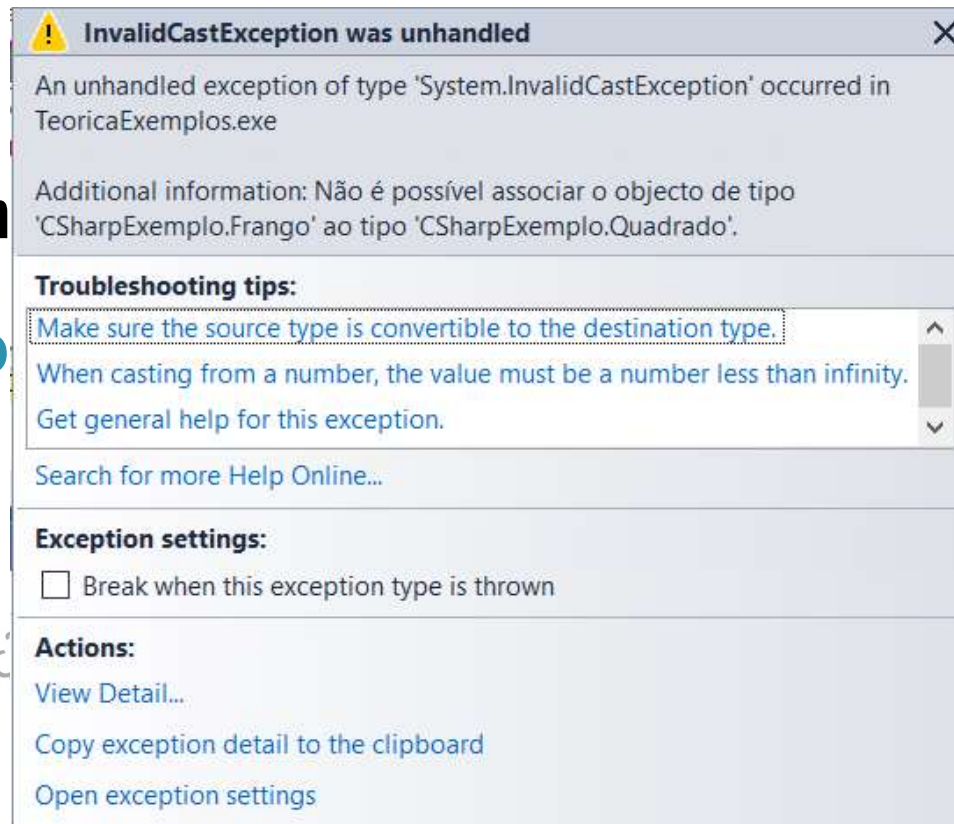
```
object meuObjecto= new Frango("NoaFrango");  
Quadrado quadrado = (Quadrado)meuObjecto;
```



C#: Será que isto compila?

object m
Quadrado

NoaFrango");
euObjecto;



Palavra chave *as* em C#

- Os *cast* explícitos são avaliados em *runtime* e não em tempo de compilação
- **as** tenta converter um objeto para um tipo específico e retorna ***null*** se a conversão falhar.

```
object meuObjecto = new Frango("FrangoObjecto");  
Quadrado quadrado = meuObjecto as Quadrado;  
if (quadrado==null)  
{  
    Console.WriteLine("O objecto não é um quadrado...");  
}
```

Palavra-chave *is* em C#

- *is* permite de forma rápida determinar se um determinado **objecto** é compatível com um **tipo** e devolve *false* caso não sejam

```
object meuObjecto = new Frango("FrangoObjecto");  
Quadrado quadrado = meuObjecto as Quadrado;  
if (quadrado is Frango)  
{  
    Console.WriteLine("O objecto é um Frango...");  
}
```

Programação Web

C#: Manipulação de Exceções

Departamento de Engenharia Informática e de Sistemas
Instituto Superior de Engenharia de Coimbra/Instituto Politécnico de Coimbra



C#: Manipulação de Exceções

- ***try***
- ***catch***
- ***finally***
- ***throw***

```
try
{
    // instruções que podem causar exceção
}
catch( ExceptionName e1 )
{
    // Código que manipula a exceção
}
catch( ExceptionName e2 )
{
    // Código que manipula a exceção
}
catch( ExceptionName eN )
{
    // Código que manipula a exceção
}
finally
{
    // Código a ser executado
}
```

C#: Manipulação de Exceções

■ E o Throw ?

- ***throw***
- ***throw ex***

```
try
{
    //algum código
}
catch
{
    throw;
}
```

```
try
{
    //algum código
}
catch (Exception ex)
{
    throw ex;
}
```

- ***throw*** - quando se utiliza o ***throw*** passamos a mesma exceção “para a frente” e com isso outro trecho de código pode capturá-la e tratar essa exceção original retendo todas as informações necessárias com o stack trace.
- ***throw ex*** - quando se utiliza o ***throw ex***, para-se a exceção ali e após se fazer alguma operação, é lançada outra exceção a partir desse ponto. Com isso, a informação de onde veio a exceção original é perdida, como se tivesse ocorrido uma nova exceção.
- De preferência devemos usar a primeira opção.

C# Manipulação de Exceções - throw

```
private int idade;
public int Idade { get { return idade; }
                  set {
                      if (value < 10 || value > 30)
                          throw new Exception("Idade Inválida!");
                      else
                          idade = value;
                  }
}
```

```
Aluno aluno = new Aluno(12342, "NomeAluno", Tipo.Normal);
try
{
    aluno.Idade = 50;
} catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

C# Manipulação de Exceções – Ex1

```
Aluno al = null;
try {
    Console.WriteLine(al.ToString());
}
catch (ArgumentNullException ex) {
    Console.WriteLine("ArgumentNullException: " + ex.Message);
}
catch (ArgumentException ex) {
    Console.WriteLine("ArgumentException: " + ex.Message);
}
catch (Exception ex) {
    Console.WriteLine("Exception: " + ex.Message);
}
finally {
    Console.WriteLine("Executa sempre...");
}
```

C# Manipulação de Exceções – Ex1

```
class DivNumeros
{
    int result;
    DivNumeros() {
        result = 0;
    }
    public void Divisao(int num1, int num2)
    {
        try { result = num1 / num2;
        }
        catch (DivideByZeroException e)
        { Console.WriteLine("{0}", e);
        }
        finally {
            Console.WriteLine("Resultado: {0}", result);
        }
    }
}
```

@JB

O que aconteceria se se fizesse esta chamada?



```
static void Main(string[] args)
{
    DivNumeros d = new DivNumeros();
    d.Divisao(25, 0);
    Console.ReadKey();
}
```



Iria tentar executar uma divisão por Zero
O que é impossível e como tal geraria uma exceção

C# Manipulação de Exceções – Ex2

```
public class Idade
{
    int idade = 12;

    public void mostraIdade()
    {
        if (idade < 18)
        {
            throw (new MaiorIdadeException("Idade inferior a 18 anos!"));
        }
        else
        {
            Console.WriteLine("Idade: {0}", idade);
        }
    }
}
```

```
public class MaiorIdadeException : Exception
{
    public MaiorIdadeException(string message) : base(message)
    { }
}
```

C# Manipulação de Exceções – Ex2

```
class VerificaIdade
{
    static void Main(string[] args)
    {
        Idade idade = new Idade();
        try
        {
            idade.mostraIdade();
        }
        catch (MaiorIdadeException e)
        {
            Console.WriteLine("MaiorIdadeException: {0}", e.Message);
        }
        Console.ReadKey();
    }
}
```

```
public class MaiorIdadeException : Exception
{
    public MaiorIdadeException(string message) : base(message)
    { }
}
```