



Programação e Sistemas de Informação

CURSO PROFISSIONAL TÉCNICO DE
GESTÃO E PROGRAMAÇÃO DE SISTEMAS INFORMÁTICOS

Programação Estruturada

MÓDULO 3

Professor: João Martiniano

Introdução

- Neste módulo iremos abordar conteúdos que nos permitem compreender a necessidade de estruturar o código e quais as formas de o fazermos
- Conteúdos do módulo:
 - Estrutura de um projeto C#
 - *Namespaces*
 - Classes
 - *Naming guidelines*
 - Métodos
 - Comentários XML
 - Classes e métodos estáticos
 - *Variable scope*
 - *Libraries*
 - Recursividade

ESTRUTURA DE UM PROJETO

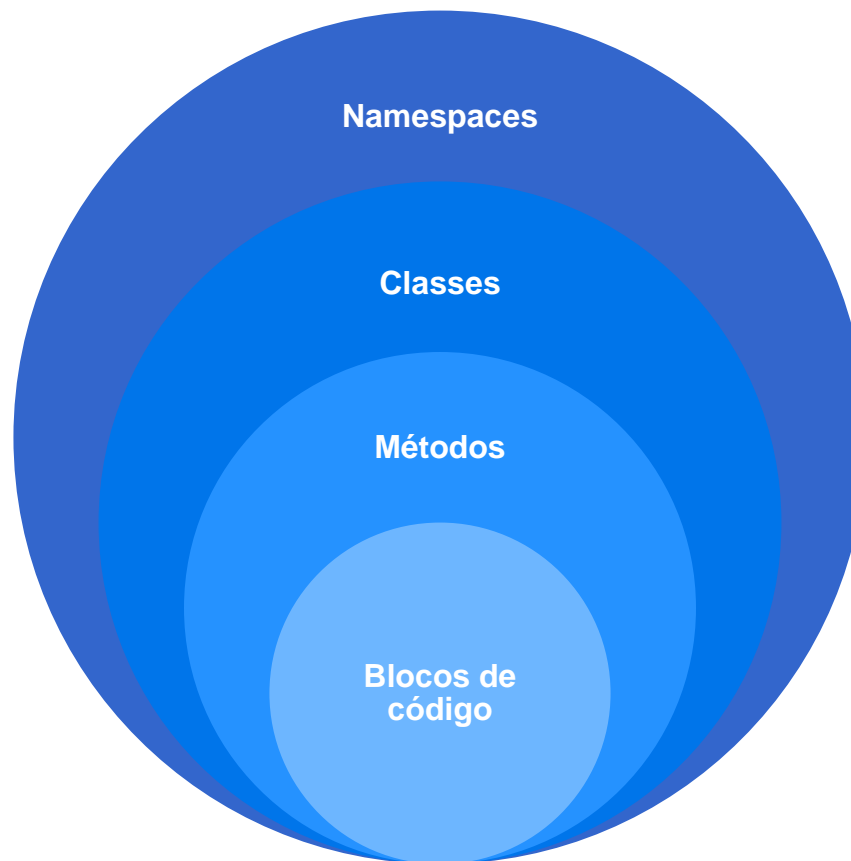
Introdução

- O trabalho no editor Visual Studio está organizado em torno de soluções e projetos
- As soluções contêm um ou mais projetos relacionados entre si
- Por exemplo, uma solução com dois projetos:
 - um projeto com um conjunto de funcionalidades: uma *library* (biblioteca)
 - um projeto com um conjunto de código para testar a *library*
- Ou seja, podemos pensar numa solução como um contentor de projetos



Introdução

- Os projetos C# estão organizados de forma hierárquica em:
 - namespaces
 - classes
 - métodos
 - blocos de código

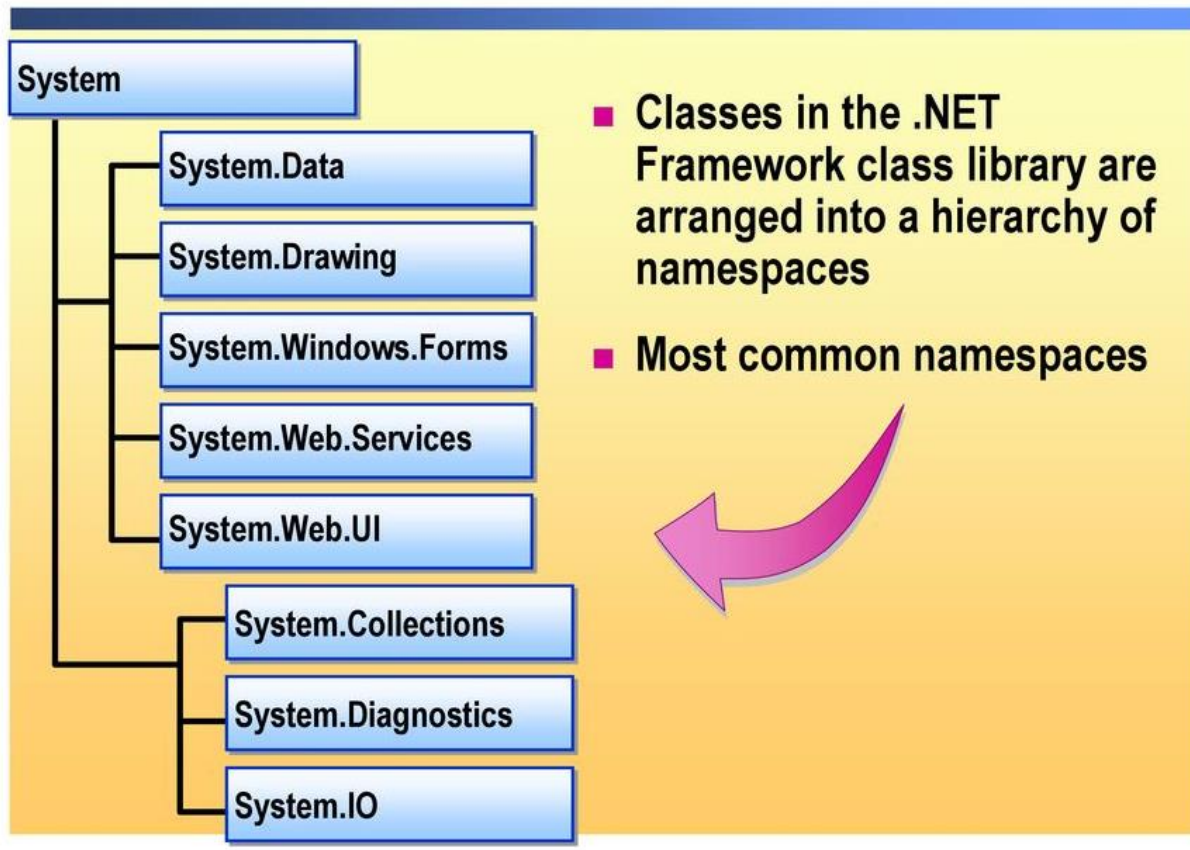


Namespaces

- Um *namespace* é uma forma de organizar e estruturar os vários elementos de um projeto
- A própria .NET Framework está organizada em *namespaces* que agrupam as várias classes e recursos
- Os componentes de um *namespace* estão organizados de forma hierárquica

Namespaces

.NET Framework Class Library



Retirado de <https://slideplayer.com/slide/12681078/>

Namespaces

- Temos vindo a utilizar *namespaces*
- Por exemplo:

System.Console.WriteLine("Olá");

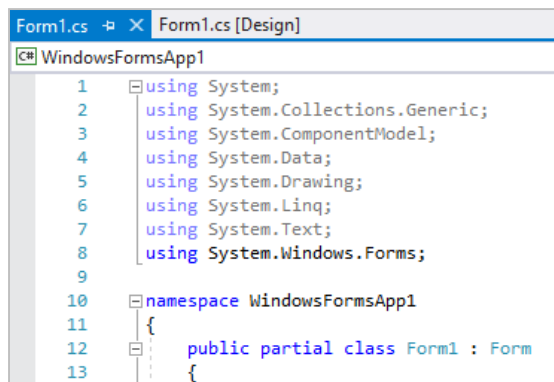
Namespace
System

Classe
Console

Método
WriteLine()

Namespaces: A diretiva `using`

- Para utilizar um namespace é utilizada a diretiva `using`
- Esta diretiva importa um *namespace* e todos os seus recursos
- Deve ser colocada no início dos ficheiros



```
Form1.cs [Design]
C# WindowsFormsApp1
1  using System;
2  using System.Collections.Generic;
3  using System.ComponentModel;
4  using System.Data;
5  using System.Drawing;
6  using System.Linq;
7  using System.Text;
8  using System.Windows.Forms;
9
10 namespace WindowsFormsApp1
11 {
12     public partial class Form1 : Form
13     {
```

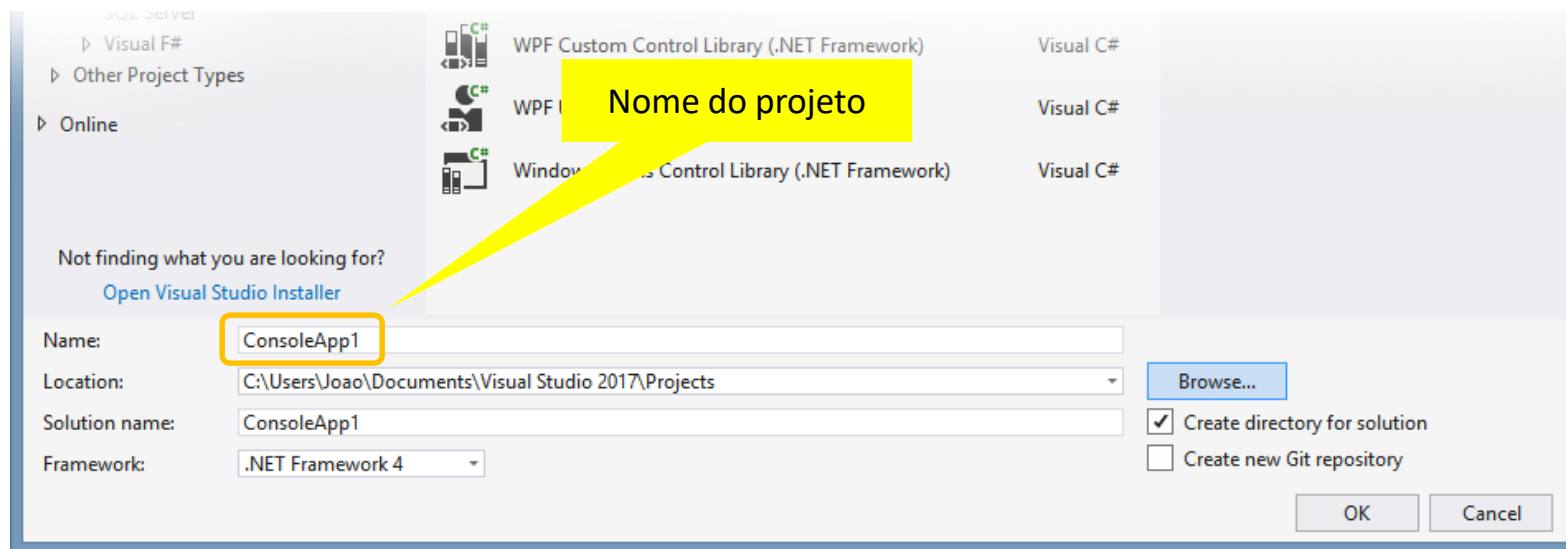
- Quando utilizamos um *namespace*, não necessitamos de prefixar os componentes com o *namespace*
- Por exemplo:
 - ao colocar a instrução `using System` no início de um ficheiro
 - em vez de `System.Console.WriteLine("Olá");`
 - podemos apenas utilizar `Console.WriteLine("Olá");`

Namespaces

- Um *namespace* pode conter zero ou mais elementos dos seguintes tipos:
 - outros *namespaces*
 - classes
 - *interfaces*
 - structs
 - enums
 - *delegates*

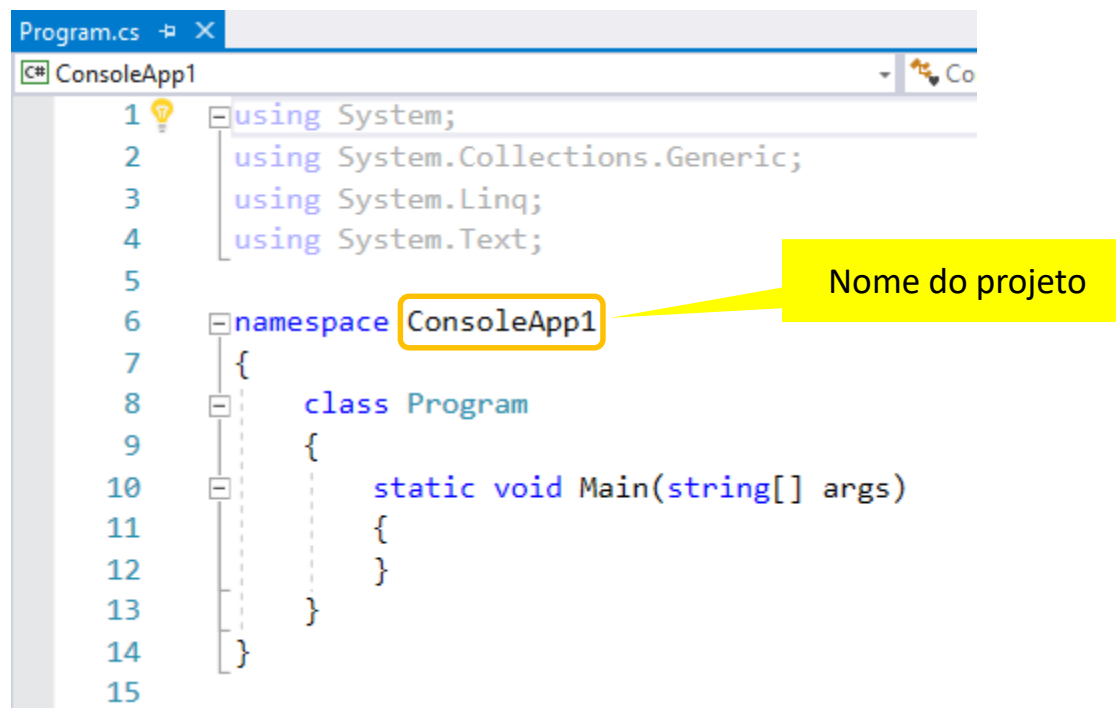
Namespaces

- Quando criamos um novo projeto, todo o conteúdo fica dentro de um *namespace* com o nome do projecto:



Namespaces

- Quando criamos um novo projeto, todo o conteúdo fica dentro de um *namespace* com o nome do projecto:



```
Program.cs - X
C# ConsoleApp1
1 using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5
6   namespace ConsoleApp1
7   {
8       class Program
9       {
10          static void Main(string[] args)
11          {
12          }
13      }
14  }
15
```

Nome do projeto

Classes

- A linguagem C# é uma linguagem de programação **orientada a objetos** (*object oriented programming language*)
- Neste paradigma o código é organizado em objetos os quais contêm:
 - dados (as propriedades e os campos)
 - operações (os métodos)
- O objectivo da programação orientada a objetos é criar programas modulares, reutilizáveis e com gestão/manutenção mais fácil
- Neste paradigma são definidas classes
- Os objetos são instâncias das classes:
 - ou seja, são definições concretas das classes

Classes

- Uma classe é composta por vários elementos, nomeadamente:
 - propriedades (properties)
 - campos (fields)
 - constantes
 - construtores
 - métodos
 - eventos
 - (e outros)
- A ideia principal a reter é que todos estes componentes estão **encapsulados** na classe
- Ou seja, apenas existem dentro da classe

Classes

- Exemplo de uma classe e respetivos componentes:

```
public class Automovel
{
    private int Velocidade;
    public string Marca { get; set; }
    public string Modelo { get; set; }
    public int Quilometragem { get; set; }

    public Automovel(string marca, string modelo, int quilometragem)
    {
        Velocidade = 0;
        Marca = marca;
        Modelo = modelo;
        Quilometragem = quilometragem;
    }

    public int Acelerar()
    {
        Velocidade += 10;
        return Velocidade;
    }
}
```

Nome da classe

Campo

Propriedades

Construtor

Método Acelerar

Naming guidelines

- Na .NET Framework existe um conjunto de *naming guidelines* (recomendações) para os identificadores dos vários componentes de um projeto
- Estas regras não são obrigatórias mas recomenda-se fortemente que os programadores as adotem nos seus programas

Capitalização

- Existem dois métodos principais para a capitalização:
 - PascalCasing: o primeiro carater de cada palavra em maiúscula
 - camelCasing:
 - o primeiro carater da primeira palavra em minúscula
 - o primeiro carater das restantes palavras em maiúscula

- Exemplos:

HtmlColor → PascalCasing

CartaoCidadao → PascalCasing

nomeCliente → camelCasing

numeroFaturaCancelada → camelCasing

Naming guidelines

- Eis algumas das *naming guidelines* para a .NET Framework:

Componente	Casing	Exemplo
Namespaces	Pascal casing	namespace Veiculos
Classes	Pascal casing	public class Automovel
Campos	Pascal casing	private int Velocidade
Propriedades	Pascal casing	public int Quilometragem { get; set; }
Métodos	Pascal casing	public int Acelerar()
Parâmetros	Camel casing	public Automovel(string marca , string modelo , int quilometragem)

- Para saber mais sobre as *naming guidelines*:
<https://docs.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines>

Naming guidelines

```
namespace Veiculos
{
    public class Automovel
    {
        private int Velocidade;
        public string Marca { get; set; }
        public string Modelo { get; set; }
        public int Quilometragem { get; set; }

        public Automovel(string marca, string modelo, int quilometragem)
        {
            Velocidade = 0;
            Marca = marca;
            Modelo = modelo;
            Quilometragem = quilometragem;
        }

        public int Acelerar()
        {
            Velocidade += 10;
            return Velocidade;
        }
    }
}
```

Namespace: Pascal casing

Classe: Pascal casing

Campo: Pascal casing

Propriedades: Pascal casing

Parâmetros: Camel casing

Método: Pascal casing

Classes: Instanciação

- Para utilizar uma classe é necessária instanciá-la, criando um novo objeto do tipo da classe
- Para tal é utilizada a instrução `new`
- Exemplo de criação de um novo objeto da classe `Automovel`:

```
static void Main(string[] args)
{
    Automovel automovel1 = new Automovel("Seat", "Leon", 15000);

    ...

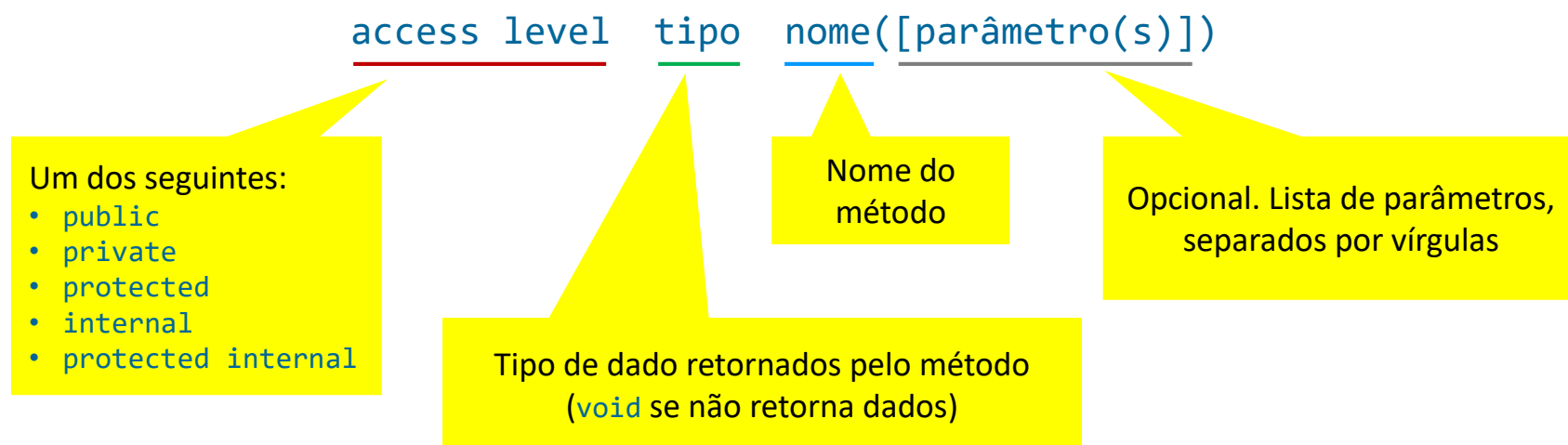
}
```

- Podem ser criadas várias instâncias de uma classe, com diferentes dados:

```
Automovel automovel2 = new Automovel("Toyota", "Aurys", 2501);
Automovel automovel3 = new Automovel("Ford", "Focus", 47200);
```

Anatomia de um método

- Um método é composto por vários componentes
- Estes componentes formam a **assinatura do método**



- Exemplo:

public string MostrarApelido(string nome)

Anatomia de um método

- Exemplo 1: método que não retorna dados e não recebe parâmetros

```
public void Desenhar()
```

- Exemplo 2: método que não retorna dados e recebe 2 parâmetros (do tipo `int`)

```
public void Deslocar(int distanciaHorizontal, int distanciaVertical)
```

- Exemplo 3: método que retorna um número inteiro e recebe 2 parâmetros (do tipo `int`)

```
public int Somar(int a, int b)
```

Anatomia de um método: Parâmetros

- Os parâmetros permitem enviar dados para dentro de um método
- Um método pode ter zero ou vários parâmetros

```
class OperacoesAritmeticas
{
    public int Somar(int a, int b)
    {
        int resultado;

        resultado = a + b;

        return resultado;
    }
}
```

Parâmetros

```
OperacoesAritmeticas op = new OperacoesAritmeticas();
int c;

c = op.Somar(2, 2);
```

Argumentos: os valores concretos
que são atribuídos aos parâmetros

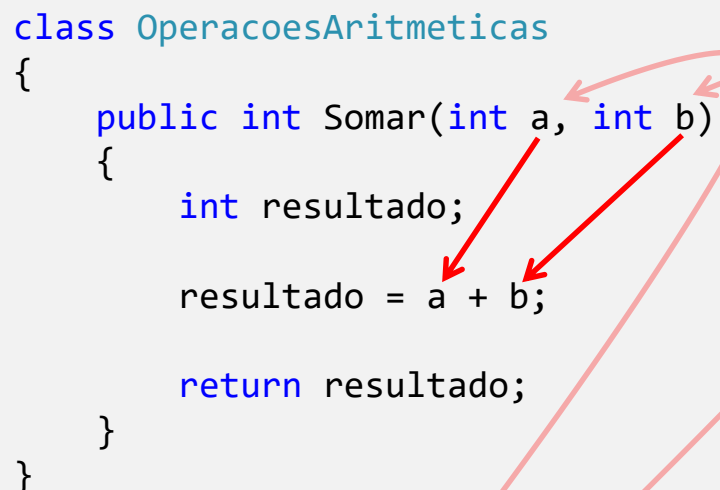
Anatomia de um método: Parâmetros

- Dentro de um método, os parâmetros funcionam como **variáveis locais**
- Ou seja, são variáveis, que existem apenas dentro do método

```
class OperacoesAritmeticas
{
    public int Somar(int a, int b)
    {
        int resultado;

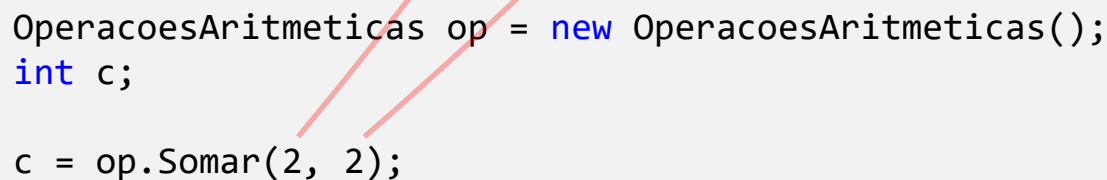
        resultado = a + b;

        return resultado;
    }
}
```



```
OperacoesAritmeticas op = new OperacoesAritmeticas();
int c;

c = op.Somar(2, 2);
```



Anatomia de um método: Retorno de valor

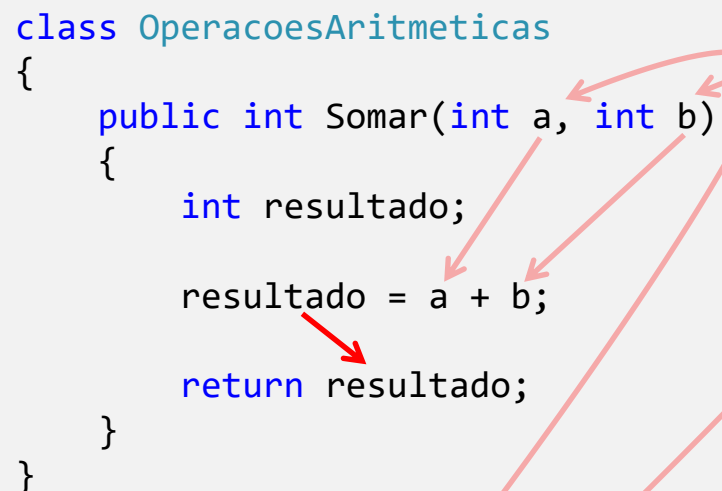
- A instrução `return` termina a execução de um método
- Permite também **retornar ou devolver** um valor
- Este valor chama-se resultado do método ou o valor de retorno
- Sintaxe:

`return [expressão];`

Anatomia de um método: Retorno de valor

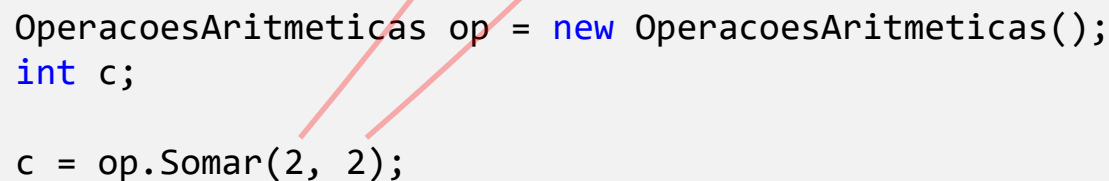
- Neste exemplo, o método `Somar()` retorna o valor da variável `resultado`

```
class OperacoesAritmeticas
{
    public int Somar(int a, int b)
    {
        int resultado;
        resultado = a + b;
        return resultado;
    }
}
```



```
OperacoesAritmeticas op = new OperacoesAritmeticas();
int c;

c = op.Somar(2, 2);
```



Anatomia de um método: Retorno de valor

- O valor retornado pelo método `Somar()` é então atribuído à variável `c`

```
class OperacoesAritmeticas
{
    public int Somar(int a, int b)
    {
        int resultado;
        resultado = a + b;
        return resultado;
    }
}
```

```
OperacoesAritmeticas op = new OperacoesAritmeticas();
int c;
c = op.Somar(2, 2);
```

Anatomia de um método: Passagem por valor e por referência

- Normalmente os parâmetros de um método são passados por valor (*value parameters*)
- Ou seja:
 - é efetuada uma cópia dos dados que são passados por parâmetro
 - os dados originais não podem ser alterados

Anatomia de um método: Passagem por valor e por referência

- Exemplo:

```
class Variaveis
{
    public void MudarNome(string nome)
    {
        nome = "Heisenberg";
    }
}
```

```
Variaveis v = new Variaveis();
string n = "Walter White";

Console.WriteLine(n);
v.MudarNome(n);
Console.WriteLine(n);
```

- Qual é o resultado, no ecrã, do código acima?

Anatomia de um método: Passagem por valor e por referência

- Exemplo:

```
class Variaveis
{
    public void MudarNome(string nome)
    {
        nome = "Heisenberg";
    }
}
```

```
Variaveis v = new Variaveis();
string n = "Walter White";
```

```
Console.WriteLine(n);
v.MudarNome(n);
Console.WriteLine(n);
```

- Qual é o resultado, no ecrã, do código acima?

```
Walter White
Walter White
```

Anatomia de um método: Passagem por valor e por referência

- Existem situações, no entanto, em que é necessário modificar o valor original de um ou mais parâmetros
- Por uma questão de rapidez e/ou quantidade de memória ocupada:
 - se se estiverem a passar dados com alguma dimensão, é ocupada mais memória
 - o processo de cópia diminui a rapidez de execução do programa
- Neste caso, os parâmetros devem ser passados por referência (*reference parameters*)
- Ou seja, apenas é passada uma referência ao valor original do parâmetro e quaisquer operações dentro do método irão modificar o valor do parâmetro, fora do método
- Para efetuar a passagem por referência deve ser incluída a *keyword* **ref** na definição do método e ao chamar o método

Anatomia de um método: Passagem por valor e por referência

- Exemplificando com o código anterior:

```
class Variaveis
{
    public void MudarNome(ref string nome)
    {
        nome = "Heisenberg";
    }
}
```

```
Variaveis v = new Variaveis();
string n = "Walter White";

Console.WriteLine(n);
v.MudarNome(ref n);
Console.WriteLine(n);
```

- O resultado do código acima é:

```
Walter White
Heisenberg
```

Anatomia de um método: *Output parameters*

- Um problema que ocorre com parâmetros passados por referência, utilizando a keyword `ref` é que estes necessitam de ser inicializados
- Por vezes é conveniente passar uma variável por parâmetro, que não foi inicializada
- Nestes casos é utilizada a keyword `out`
- É também útil nos casos em que é necessário devolver mais do que um valor
- O método `TryParse()` é um bom exemplo de utilização de *output parameters*:
 - o método verifica se é possível um valor de um tipo, para outro tipo
 - se não for possível, o método devolve o valor `false`
 - se for possível, devolve o valor `true` e retorna o valor convertido para o tipo desejado, através de um parâmetro

Anatomia de um método: *Output parameters*

- No seguinte exemplo, se o método `int.TryParse()` determinar que é possível converter a string `n` para o tipo `int`, o valor convertido é atribuído à variável `x`:

```
string n = "544";  
int x;  
  
if (int.TryParse(n, out x))  
{  
    // fazer qualquer coisa  
}
```

Anatomia de um método: *Output parameters*

- No seguinte exemplo, o método `SomarEspecial()` exemplifica como definir um *output parameter*:
 - se os parâmetros `a` e `b` forem superiores a `10`, o método retorna `true` e o resultado da soma é colocado no parâmetro `c`
 - caso contrário, o método retorna `false` e é atribuído `-1` ao parâmetro `c`

```
class OperacoesAritmeticas
{
    public bool SomarEspecial(int a, int b, out int c)
    {
        if ((a > 10) && (b > 10))
        {
            c = a + b;
            return true;
        }
        else
        {
            c = -1;
            return false;
        }
    }
}
```

Anatomia de um método: Parâmetros opcionais

- Parâmetros opcionais são parâmetros para os quais são definidos valores por defeito
- Caso não seja atribuído um argumento, o parâmetro assume o valor por defeito
- Os parâmetros opcionais devem sempre ser definidos em último lugar

Anatomia de um método: Parâmetros opcionais

- No seguinte exemplo o parâmetro `b` é opcional
- Caso não seja fornecido um valor, o parâmetro assume o valor `0`

```
class OperacoesAritmeticas
{
    public int SomarOpcional(int a, int b = 0)
    {
        return a + b;
    }
}
```

```
OperacoesAritmeticas op = new OperacoesAritmeticas();
int r;

r = op.SomarOpcional(2, 8);
Console.WriteLine(r);

r = op.SomarOpcional(2);
Console.WriteLine(r);
```

Anatomia de um método: Parâmetros opcionais

- O resultado do código anterior é:

```
10  
2
```

Comentários XML

- Para além dos comentários tradicionais (`///` e `/* */`) existe um terceiro tipo de comentários: os comentários XML (ou *XML comments*)
- Este tipo de comentários servem duas funções:
 - permitem gerar documentação sobre o código
 - são integrados diretamente no sistema **Intellisense** do editor Visual Studio ajudando o programador
- Os comentários XML começam com os caracteres `///`
- Contêm *tags* as quais contêm os comentários propriamente ditos
- Os comentários XML são aplicados aos tipos e membros definidos pelo utilizador: classes, métodos, propriedades, campos, enumerações, etc.

Comentários XML

- No seguinte exemplo são acrescentados comentários XML para descrever a classe `OperacoesAritmeticas` e o método `Somar()`:

```
/// <summary>
/// Implementa operações aritméticas básicas.
/// </summary>
class OperacoesAritmeticas
{
    /// <summary>
    /// Efetua a soma de dois números inteiros.
    /// </summary>
    public int Somar(int a, int b)
    {
        int resultado;

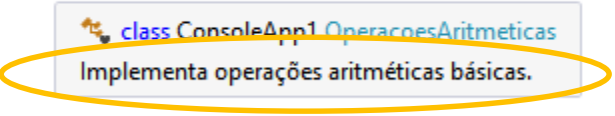
        resultado = a + b;

        return resultado;
    }
}
```

Comentários XML

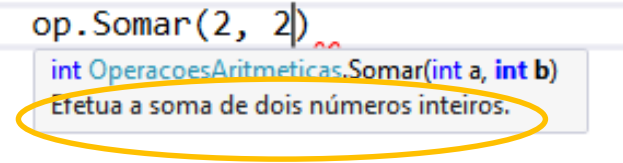
- Ao utilizarmos elementos com comentários XML, estes ficam integrados no mecanismo **Intellisense**
- Por exemplo, ao utilizar a classe `OperacoesAritmeticas`, se o utilizador colocar o cursor por cima do nome da classe surge a descrição desta, definida anteriormente:

```
OperacoesAritmeticas op = new OperacoesAritmeticas();
```



class ConsoleApp1.OperacoesAritmeticas
Implementa operações aritméticas básicas.

- Ao utilizar o método `Somar()` surge, também, a descrição do mesmo:



```
op.Somar(2, 2)
```

int OperacoesAritmeticas.Somar(int a, int b)
Efetua a soma de dois números inteiros.

Comentários XML

- Principais *tags* que podem ser utilizadas nos comentários XML:

- `<summary>` Adiciona um sumário acerca do tipo/membro. Esta tag deve veicular a informação mais importante.
- `<remarks>` Informação adicional, que complementa a informação contida na tag `<summary>`
- `<returns>` Descreve o valor de retorno de um método
- `<param>` Descrição dos parâmetros de um método

- Existem mais *tags* disponíveis para utilização, as quais permitem documentar o código com grande detalhe

Comentários XML

- Utilizando as *tags* descritas anteriormente, podemos acrescentar informações à classe `OperacoesAritmeticas`:

```
/// <summary>
/// Implementa operações aritméticas básicas.
/// </summary>
class OperacoesAritmeticas
{
    /// <summary>
    /// Efetua a soma de dois números inteiros.
    /// </summary>
    /// <param name="a">Número inteiro.</param>
    /// <param name="b">Número inteiro.</param>
    /// <returns>A soma de dois números inteiros.</returns>
    public int Somar(int a, int b)
    {
        int resultado;

        resultado = a + b;

        return resultado;
    }
}
```

Classes e métodos estáticos

- Uma classe estática é uma classe que não pode ser instanciada
- Uma classe estática apenas pode conter membros estáticos
- Os membros de uma classe estática são acedidos utilizando o nome da classe
- Para especificar que uma classe ou membro de uma classe é estatático, é utilizada a *keyword* `static`
- A classe `Math` constitui um exemplo de uma classe estática

Classes e métodos estáticos

- A classe `OperacoesAritmeticas`, dada a sua natureza, é um bom exemplo de uma classe estática:
- Convertendo a classe e os respetivos membros para `static`...

```
public static class OperacoesAritmeticas
{
    public static int Somar(int a, int b)
    {
        int resultado;

        resultado = a + b;

        return resultado;
    }
}
```

- ... os respetivos métodos podem ser executados diretamente, sem que seja necessário criar uma nova instância:

```
int c = OperacoesAritmeticas.Somar(2, 2);
```

VARIABLE SCOPE

Variable scope: Introdução

- As variáveis, bem como outros componentes de um programa (como os métodos) têm um alcance diferente, consoante onde são declarados ou definidos
- A isto chama-se o ***scope*** (pode ser traduzido como o alcance ou âmbito)
- Ou seja, uma variável está acessível num determinado *scope*
- Na linguagem C# existem os seguintes níveis de *scope* para as variáveis:

Class level scope

Method level scope

Block level scope

Variable scope: Class level scope

- Um campo de uma classe é uma variável cujo *scope* é toda a classe
- Qualquer método da classe pode aceder à variável para ler e modificar o seu valor
- No seguinte exemplo, a classe `Pessoa` possui o campo `Nome`:

```
class Pessoa
{
    private string Nome;

    public void MostrarNome()
    {
        Console.WriteLine("O nome desta pessoa é {0}", Nome);
    }

    public void LimparNome()
    {
        Nome = "";
    }
}
```

Scope do campo `Nome`

Este método lê o valor de `Nome`

Este método modifica o valor de `Nome`

Scope do campo `Nome`

Variable scope: Method level scope

- As variáveis declaradas dentro dos métodos, apenas existem dentro destes
- Significa que:
 - assim que um método termina, termina o tempo de vida da variável
 - a variável não pode ser acedida fora do método onde foi declarada

Variable scope: Method level scope

- No seguinte exemplo é declarada a variável `x` dentro do método `A()`
- O método `B()` tenta aceder à variável `x`, o que irá provocar um erro...
- ... porque a variável `x` não existe no método `B()`

```
class Variaveis
{
    public void A()
    {
        int x = 33;
        Console.WriteLine("O valor de x é {0}", x);
    }

    public void B()
    {
        Console.WriteLine("O valor de x é {0}", x);
    }
}
```

Scope
da
variável
`x`

Provoca um erro

Variable scope: Method level scope

- Ambos os métodos podem conter uma variável `x`
- São variáveis diferentes, com *scope* diferente

```
class Variaveis
{
    public void A()
    {
        int x = 33;
        Console.WriteLine("O valor de x é {0}", x);
    }

    public void B()
    {
        int x = 1;
        Console.WriteLine("O valor de x é {0}", x);
    }
}
```

Scope da
variável `x`
do
método
`A()`

Scope da
variável `x`
do
método
`B()`

Variable scope: Method level scope

- Os parâmetros são como variáveis, cujo *scope* é o método
- Assim sendo, num método não pode existir uma variável local e um parâmetro com um nome idêntico
- O seguinte exemplo demonstra esta situação:

```
class Variaveis
{
    public void A(int x)
    {
        int x = 33;
        Console.WriteLine("O valor de x é {0}", x);
    }
}
```

Este código origina um erro

Variable scope: Method level scope

- Surpreendentemente, um campo e uma variável local podem ter o mesmo nome e não se sobrepõem:

```
class Variaveis
{
    private int x = 100;

    public void A()
    {
        int x = 33;

        Console.WriteLine("O valor de x é {0}", x);
    }
}
```

Sem problemas: o campo `x` e a variável local `x` são consideradas entidades diferentes

- Neste caso, se executarmos o método `A()` qual será o resultado no ecrã?

Variable scope: Method level scope

- Surpreendentemente, um campo e uma variável local podem ter o mesmo nome e não se sobrepõem:

```
class Variaveis
{
    private int x = 100;

    public void A()
    {
        int x = 33;

        Console.WriteLine("O valor de x é {0}", x);
    }
}
```

Sem problemas: o campo `x` e a variável local `x` são consideradas entidades diferentes

- Neste caso, se executarmos o método `A()` qual será o resultado no ecrã?

O valor de x é 33

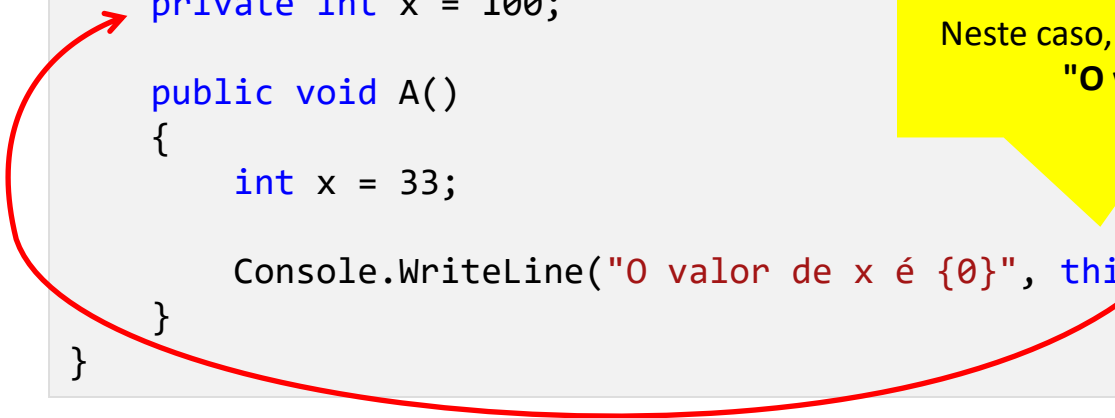
Variable scope: Method level scope

- Então como referenciar num método, o valor do campo `x`, quando há uma variável local com o mesmo nome?
- Utilizando a *keyword* `this`
- Esta *keyword* permite referenciar a instância da classe, onde é utilizada
- Ou seja:
 - ao utilizar `this.x` está-se a referenciar o campo `x` da classe `Variaveis`
 - e não a variável local `x` do método `A()`

```
class Variaveis
{
    private int x = 100;

    public void A()
    {
        int x = 33;

        Console.WriteLine("O valor de x é {0}", this.x);
    }
}
```



Neste caso, o resultado no ecrã seria
"O valor de x é 100"

Variable scope: Block level scope

- Uma variável com *block level scope* é uma variável declarada dentro de um bloco de código
- São variáveis que são declaradas dentro de:
 - instruções repetitivas (`while`, `do...while`, `for`, `foreach`)
 - instruções condicionais (`if...else`, `switch`)
 - outros blocos de código

Variable scope: Block level scope

- No exemplo seguinte:
 - a variável `x` é declarada dentro do ciclo `while`
 - a variável `mensagem` é declarada dentro da instrução `if`

```
int i = 1;

while (i <= 5)
{
    int x = 2;

    Console.WriteLine("{0} x {1} = {2}", i, x, i * x);
    ++i;
}

if (i == 5)
{
    string mensagem = "Fim do ciclo while";

    Console.WriteLine(mensagem);
}
```

Scope da
variável `x`
do ciclo
`while`

Scope da
variável
`mensagem`
da
instrução
`if`

Variable scope: Block level scope

- As variáveis com *block level scope* apenas existem dentro do bloco onde são declaradas, não podendo ser utilizadas fora do respetivo bloco:

```
int i = 1;

while (i <= 5)
{
    int x = 2;

    Console.WriteLine("{0} x {1} = {2}", i, x, i * x);
    ++i;
}

Console.WriteLine(x);

if (i == 5)
{
    string mensagem = "Fim do ciclo while";

    Console.WriteLine(mensagem);
}

Console.WriteLine(mensagem);
```

Este código origina um erro

Este código origina um erro

Variable scope: Block level scope

- Há que ter em atenção que uma variável com *block level scope* não pode ter o mesmo nome de uma variável local:

```
int x = 100;
int i = 1;

while (i <= 5)
{
    int x = 2;
    Console.WriteLine("{0} x {1} = {2}", i, x, i * x);
    ++i;
}
```

Este código origina um erro

Variable scope: Block level scope

- É habitual a declaração de variáveis, em ciclos **for**, da seguinte forma:

```
for (int i = 1; i <= 5; ++i)
{
    Console.WriteLine(i);
}
```

- Neste caso, o *scope* da variável **i** é o ciclo **for**

Variable scope: Block level scope

- Por vezes, em situações específicas, são criados blocos de código anónimos (*anonymous code blocks*)
- Dentro destes blocos podem ser declaradas variáveis cujo *scope* é o bloco onde são declaradas

Scope das
variáveis
`nome`
e
`apelido`

```
{  
    string nome = "Harry", apelido = "Potter";  
    Console.WriteLine("{0} {1}", nome, apelido);  
}
```

Console.WriteLine(nome);

Este código origina um erro

- Não é possível utilizar as variáveis fora do bloco de código anónimo

| *LIBRARIES*

Libraries

- Em qualquer linguagem de programação existe o conceito de *library*
- Ou seja, às instruções nucleares da própria linguagem são acrescentadas funcionalidades na forma de recursos, com diversos fins
- Podemos pensar na .NET Framework como sendo como uma gigantesca *library* (é mais do que isso)
- A .NET Framework fornece uma impressionante quantidade de recursos e funcionalidades aos programadores
- E essa quantidade está em constante crescimento: a cada nova versão da .NET Framework são acrescentadas mais funcionalidades, recursos e tecnologias

Libraries

- Algumas classes úteis:

Namespace	Classe	Descrição
System	Math	Fornece métodos e constantes para efetuar operações matemáticas.
System	Environment	Fornece dados e formas de manipular o ambiente/plataforma de computação.
System	OperatingSystem	Contém dados sobre o sistema operativo.
System	Random	Gerador de números pseudo-aleatórios.
System	DateTime (<i>struct</i>)	Permite armazenar e trabalhar com datas/horas. Não é uma classe, é uma <i>struct</i> .

System.Math

- O seguinte exemplo demonstra algumas funcionalidades/recursos da classe `Math`, nomeadamente: raiz quadrada, potenciação, arredondamento, etc.

```
Console.WriteLine("PI = {0}", System.Math.PI);  
Console.WriteLine("Raiz quadrada de 25 = {0}", System.Math.Sqrt(25));  
Console.WriteLine("2^6 = {0}", System.Math.Pow(2, 6));  
Console.WriteLine("Arredondar 2,6 para número inteiro = {0}",  
System.Math.Round(2.6, 0));  
Console.WriteLine("Arredondar 18,477 para número com 1 casa decimal =  
{0}", System.Math.Round(18.477, 1));  
Console.WriteLine("Truncar número 2.3 = {0}", Math.Truncate(2.3));  
Console.WriteLine("Truncar número 2.9 = {0}", Math.Truncate(2.9));
```


System.Environment

- O seguinte exemplo demonstra como obter informações sobre a plataforma informática na qual uma aplicação é executada:

```
Console.WriteLine("Dados de environment\n");
Console.WriteLine("Informações do computador\n-----");
Console.WriteLine("Nome do computador: {0}", System.Environment.MachineName);
Console.WriteLine("Nº de processadores: {0}", System.Environment.ProcessorCount);
Console.WriteLine("");
Console.WriteLine("Informações do sistema operativo\n-----
--");
Console.WriteLine("Sistema operativo de 64 bits: {0}",
System.Environment.Is64BitOperatingSystem);
Console.WriteLine("Versão do sistema operativo: {0}",
System.Environment.OSVersion);
Console.WriteLine("Username: {0}", System.Environment.UserName);
Console.WriteLine("Está em curso operação de shutdown? {0}",
System.Environment.HasShutdownStarted);
Console.WriteLine("Diretoria de sistema: {0}", System.Environment.SystemDirectory);
Console.WriteLine("Diretoria desta aplicação: {0}",
System.Environment.CurrentDirectory);
```

System.Environment

- A classe `System.Environment` contém o método `Exit()`
- Este método termina uma aplicação de consola, com um determinado *exit code*
- O *exit code* é um código numérico que sinaliza ao sistema operativo que:
 - a aplicação terminou com sucesso (*exit code* = 0)
 - a aplicação terminou com um erro (*exit code* diferente de 0)
- Exemplo 1: sinalizar que a aplicação terminou com sucesso

```
System.Environment.Exit(0);
```

- Exemplo 2: sinalizar que a aplicação terminou com um erro

```
System.Environment.Exit(1);
```

System.OperatingSystem

- O seguinte exemplo demonstra como obter dados sobre o sistema operativo:

```
// O objeto sistemaOperativo contém os dados do sistema operativo
OperatingSystem sistemaOperativo = Environment.OSVersion;

Console.WriteLine("Dados deste Sistema Operativo:\n");
Console.WriteLine("Plataforma: {0:G}", sistemaOperativo.Platform);
Console.WriteLine("Version String: {0}", sistemaOperativo.VersionString);
Console.WriteLine("Version Information:");
Console.WriteLine("    Major: {0}", sistemaOperativo.Version.Major);
Console.WriteLine("    Minor: {0}", sistemaOperativo.Version.Minor);
Console.WriteLine("Service Pack: '{0}'", sistemaOperativo.ServicePack);
```

System.Random

- O seguinte exemplo demonstra como gerar três tipos de números pseudo-aleatórios (não verdadeiramente aleatórios):

```
Random rnd = new System.Random();

// Número inteiro aleatório entre 1 e 45
Console.WriteLine("Número inteiro aleatório entre 1 e 45: {0}", rnd.Next(1, 46));

// Número real aleatório entre 0.0 e 0.9
Console.WriteLine("Número real aleatório >= 0.0 e < 1.0 : {0}", rnd.NextDouble());

// Número real aleatório entre 0.0 e 5.9
Console.WriteLine("Número real aleatório entre 0 e 5: {0}", rnd.NextDouble() * 6);
```

System.DateTime

- A `struct DateTime` é uma estrutura que permite armazenar e trabalhar com datas e horas
- Disponibiliza várias funcionalidades para diversas operações relacionadas com tempo

System.DateTime

Declaração

- Esta estrutura contém vários construtores, o que fornece flexibilidade para utilização em várias situações
- Os seguintes exemplos demonstram algumas das formas possíveis de declarar uma variável deste tipo

- Exemplo 1: Declarar uma variável com a data de 25 de janeiro de 2020

```
DateTime data1 = new DateTime(2020, 1, 25);
```

- Exemplo 2: Declarar uma variável com a data de 25 de janeiro de 2020, 09h05m10s

```
DateTime data2 = new DateTime(2020, 1, 25, 9, 5, 10);
```

System.DateTime

Declaração

- Para além disso, as propriedades **Today** e **Now** retornam:
 - **Today**: a data atual (as horas são ignoradas)
 - **Now**: a data/hora atual
- Estas propriedades também podem ser utilizadas para atribuir um valor inicial a uma variável deste tipo
- Exemplo 3: Declarar uma variável com a data atual

```
DateTime data3 = DateTime.Today;
```

- Exemplo 4: Declarar uma variável com a data/hora atual

```
DateTime data4 = DateTime.Now;
```

System.DateTime

Propriedades

- A estrutura contém várias propriedades que permitem aceder aos valores individuais da data/hora

```
DateTime data1 = new DateTime(2020, 1, 25);  
DateTime data2 = new DateTime(2020, 1, 25, 9, 5, 10);  
  
int ano = data1.Year; // Obter o ano (2020)  
int mes = data1.Month; // Obter o mês (1)  
int dia = data1.Day; // Obter o dia (25)  
  
// Obter o dia da semana (em número: 6)  
int diaSemanaNumero = (int)data1.DayOfWeek;  
// Obter o dia da semana (em string: Saturday)  
string diaSemana = data1.DayOfWeek.ToString();  
  
int hora = data2.Hour; // Obter as horas (9)  
int minutos = data2.Minute; // Obter os minutos (5)  
int segundos = data2.Second; // Obter os segundos (10)
```


System.DateTime

Parsing

- É por vezes necessário efetuar *parsing* de uma data/hora proveniente de uma string
- Para tal utilizam-se os métodos `Parse()` e `TryParse()`
- Exemplos:

```
// 15/04/2009
DateTime data = DateTime.Parse("15 de abril de 2009");

// 02/03/1985
data = DateTime.Parse("1985-03-02");

// 16:12
data6 = DateTime.Parse("16:12");
```

System.DateTime

Output

- Existem várias formas de efetuar output de uma variável do tipo `DateTime`
- Uma maneira consiste em utilizar alguns métodos da `struct`:

```
DateTime data2 = new DateTime(2020, 1, 25, 9, 5, 10);

// Data/hora completa: 25/01/2020 09:05:10
Console.WriteLine("\n\nToString(): {0}", data2.ToString());

// Data em formato longo: 25 de janeiro de 2020
Console.WriteLine("ToLongDateString(): {0}", data2.ToLongDateString());

// Data em formato curto: 25/01/2020
Console.WriteLine("ToShortDateString(): {0}", data2.ToShortDateString());

// Horas em formato longo: 09:05:10
Console.WriteLine("ToLongTimeString(): {0}", data2.ToLongTimeString());

// Horas em formato curto: 09:05
Console.WriteLine("ToShortTimeString(): {0}", data2.ToShortTimeString());
```

System.DateTime

Output

- Outra forma de efetuar output consiste, com o método `ToString()`, utilizar caracteres que têm um significado especial (*specifiers*) para que o output fique no formato desejado:

Specifier	Significado
d	Dia do mês (1 a 31)
dd	Dia do mês (01 a 31)
M	Mês (1 a 12)
MM	Mês (01 a 12)
MMMM	Nome do mês
yy	Ano (2 dígitos)
yyyy	Ano (4 dígitos)
HH	Hora (00 a 23)
mm	Minutos (00 a 59)
ss	Segundos (00 a 59)

System.DateTime

Output

- Exemplos:

```
DateTime data2 = new DateTime(2020, 1, 25, 9, 5, 10);

// 25-01-2020
Console.WriteLine(data2.ToString("dd-MM-yyyy"));

// 25/janeiro/2020
Console.WriteLine(data2.ToString("dd/MMMM/yyyy"));

// 25, janeiro de 20
Console.WriteLine(data2.ToString("dd, MMMM \"de\" yy"));

// 09h05
Console.WriteLine(data2.ToString("HH\"h\"mm"));
```

System.DateTime

Operações

- É comum necessitar de efetuar operações com datas/horas
- Para tal, utilizam-se métodos de `DateTime`
- Os seguintes métodos adicionam quantidades a uma data/hora:

`AddDays()`

`AddMonths()`

`AddYears()`

`AddHours()`

`AddMinutes()`

`AddSeconds()`

`AddMilliseconds()`

- O valor a adicionar é especificado como parâmetro em cada um dos métodos
- Para subtrair, deve-se especificar um valor negativo
- Estes métodos retornam uma nova `struct DateTime`

System.DateTime

Operações

- Exemplo:

```
DateTime data2 = new DateTime(2020, 1, 25, 9, 5, 10);

// Adicionar 5 dias: 30/01/2020
Console.WriteLine(data2.AddDays(5));

// Subtrair 30 dias: 26/12/2019
Console.WriteLine(data2.AddDays(-30));

// Adicionar 1 mês: 25/02/2020
Console.WriteLine(data2.AddMonths(1));

// Adicionar 2 anos: 25/01/2022
Console.WriteLine(data2.AddYears(2));

// Adicionar 4 horas: 13:05:10
Console.WriteLine(data2.AddHours(4));
```

| RECURSIVIDADE

Recursividade

- Um método recursivo é um método que se chama a si próprio
- Alguns problemas são mais facilmente resolvidos utilizando recursividade (tal como a implementação de determinados algoritmos matemáticos)
- No entanto: a recursividade consome mais recursos do computador
- Deve ser utilizada com ponderação

Recursividade

- Para demonstrar o conceito de recursividade analisemos o cálculo de fatorial
- Definição: o fatorial de um número n consiste no produto (multiplicação) de todos os inteiros positivos menores ou iguais a n

$$n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1$$

(caso especial: $0! = 1$)

- Exemplo:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Recursividade

- Algoritmo iterativo para o cálculo de fatorial:

```
public long Fatorial(int n)
{
    if (n == 0)
    {
        return 1;
    }

    long resultado = 1;

    for (int i = 1; i <= n; ++i)
    {
        resultado *= i;
    }

    return resultado;
}
```

Recursividade

- Algoritmo recursivo para o cálculo de fatorial:

```
public long FatorialRecursivo (long n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * FatorialRecursivo(n - 1);
    }
}
```