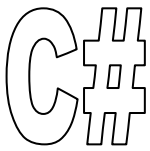
C# Guia de Referência

Autor:
Fabio Renato de Almeida

Curso C# 1.0 e 2.0

(Guia de Referência do Aluno)

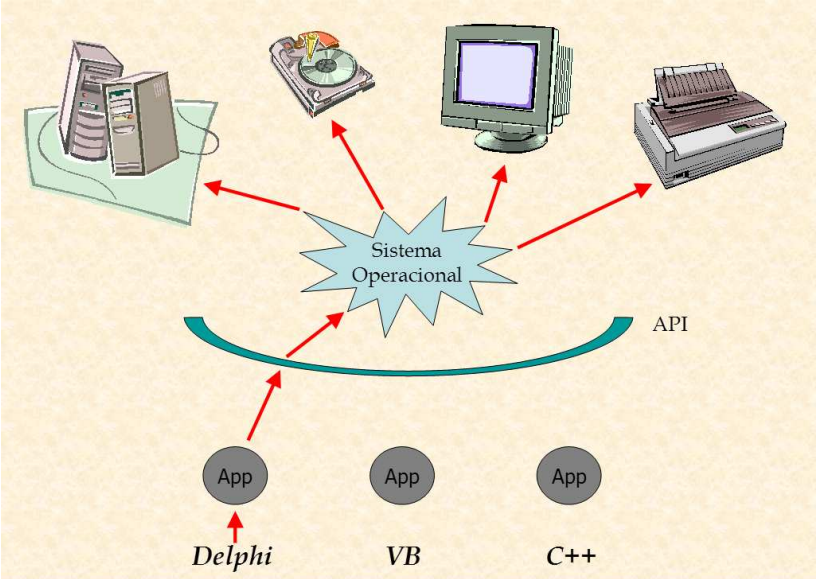


Instrutor: Fábio Renato de Almeida

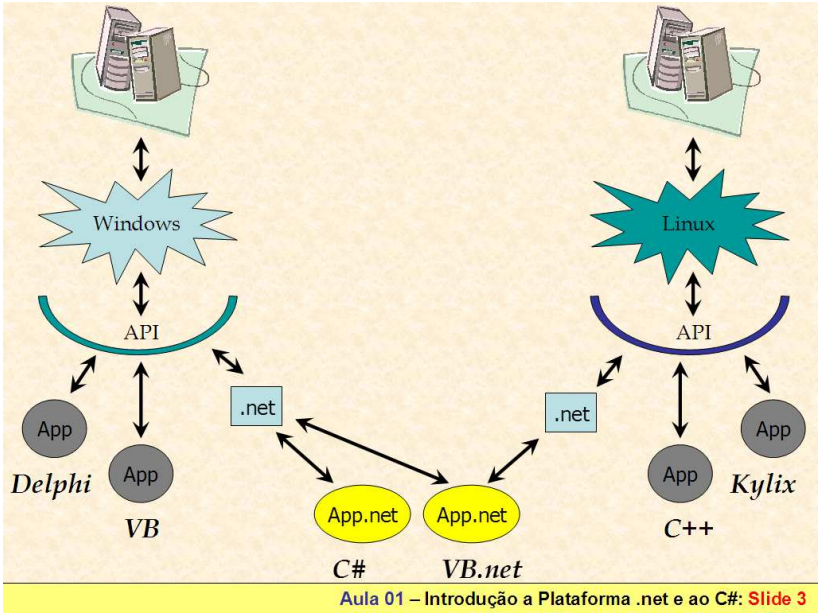
fabiorenato.net@uol.com.br

1. Introdução a Plataforma .net e ao C#.

2. Visão Geral (Sistema Operacional).



3. .net Visão Geral (Windows / Linux).



4. Por que .net Framework?

- .net (***Aplicativos em rede "network"***).
- Framework (***Infra-estrutura / Plataforma***).
- .net Framework: Plataforma projetada, desde o início, com o propósito de ser eficiente em **aplicações distribuídas**.
  - LAN (Local Area Network).
  - MAN (Metropolitan Area Network).
  - WAN (Wide Area Network / **Internet**).

5. .net Framework - Características.

- Totalmente **orientado a objeto** (OO).
  - Suporte a **classes**.
  - Mecanismo de **herança simples** e **interfaces**.

- Funções virtuais (**polimorfismo**).

➢ ***Case-sensitive***.

➢ Definição clara de ***tipos valor*** e ***tipos referência***.
- Tipos valor: **Stack** (Pilha).
- Tipos referência: **Heap** (Área gerenciada).
- Tipos valor que são declarados como um campo em um tipo referência, neste caso, também são armazenados no heap, junto com seu container (**boxed values**).
- Dados fortemente tipados (***type-safe***).
  - ♦ Não existe VARIANT... não confunda Object com Variant... mesmo um Object, contém a especificação do tipo real da instância... e seus métodos polimórficos serão devidamente invocados.

6. .net Framework - Características.

➢ CTS (**Common Type System**).
- **Int16**, **Int32**, **Int64**, **Single**, **Double**, ... todas as linguagems que tenham como alvo o .net, devem produzir código compilado baseado nesses tipos.
  - ♦ VB.net: **Dim valor As Integer** => valor (**Int32**).
  - ♦ C#: **int valor;** => valor (**Int32**).
  - ♦ Delphi: **var valor: Integer;** => valor (**Int32**).

➢ CLS (**Common Language Specification**).
- Conjunto de regras que todos os compiladores que tenham como alvo a plataforma .net devem implementar.
  - ♦ **System.Int32** é CLS.
  - ♦ **System.UInt16** (0...65535) não é CLS.
  - ♦ Diferenciação entre ***maiúsculas*** e ***minúsculas*** não é CLS.

7. .net Framework - Características.

➢ FCL (**Framework Class Library**): Extensa biblioteca de classes, organizada por espaço de nome (*forma de agrupar classes relacionadas, ainda que em assemblies diferentes*). Bom design.
- Win 3.x, 95, 98, Me, 2000, XP, 2003: **API**.
- Plataforma .net: **FCL**.
- Longhorn: **FCL**.

8. .net Framework - Características.

➢ IL (**Intermediate Language**).
- C#, Delphi for .net, VB.net... apenas a maneira como se escreve... ***o resultado final é compilado para IL***.
- Linguagem de **baixo nível**, com ***sintaxe simples***, **baseada em números ao invés de texto**, que pode ser **traduzida rapidamente para código nativo**.
- **A linguagem que o .net reconhece**.
- Todos os compiladores para .net produzem código IL.
- Independência de...
  - ♦ ***Plataforma***.
    - ⇒ Windows (Microsoft .net Framework).
    - ⇒ Linux, Mac OS X, Solaris, Windows, Unix (Mono).
  - ♦ ***Linguagem***.
    - ⇒ C#, VB.net, Delphi for .net, entre outras... (mais de 30).
    - ⇒ Todas geram código IL (Intermediate Language).
    - ⇒ A programação é feita, não para o S.O., mas para a plataforma .net.

9. .net Framework - Características.

➢ Assembly (**Montagem**).
- O assembly pode ser um **EXE** ou uma **DLL**.
- Auto-descritivo (***Metadados***).
  - ♦ Contém informações sobre os membros do assembly.
  - ♦ Contém informações sobre ele mesmo (**manifesto**).
  - ♦ Versionamento: **Fim do Inferno das DLLs (DLL Hell).**

    - ♦ ***Strong Name***.
    - ♦ Toda informaçao possível sobre o assembly está no próprio assembly.
  - Não há entradas no registro do sistema (**GUID**).
  - ***Privado*** ou ***compartilhado*** (**GAC: Global Assembly Cache**).
  - **Reflection** (Reflexão)... através dos metadados.
  - **Instalação de zero impacto (xcopy)**.

10. .net Framework - Características.

- ***Interoperabilidade*** entre linguagens.
  - **Lib1.dll** escrita em **C#.**
  - **Lib2.dll** escrita em **VB.net** herdando de **Lib1.dll**.
  - **Lib3.dll** escrita em **Delphi for .net** herdando de **Lib2.dll**.
  - **Lib4.dll** escrita em **C#** herdando de **Lib3.dll**.
  - O mecanismo de tratamento de exceções também permite que ***exceções lançadas na Lib2.dll possam ser devidamente capturadas e tratadas na Lib4.dll***.

11. .net Framework - Características.

- ***Interoperabilidade COM***.
  - Uma **classe .net wrapper** é criada para gerenciar objetos COM.
  - **A tecnologia COM é considerada ultrapassada**.
  - **O ideal é escrever as funcionalidades COM em código gerenciável**.
- ***Segurança***.
  - Possibilidade de executar um assembly sem risco de código malicioso.
  - Um assembly pode conter informações sobre ***quem ou qual grupo tem permissões para executar determinados métodos no assembly [atributos]***.
  - .**net oferece segurança baseada em código**. Windows oferece segurança baseada em roles (usuário, grupo, domínio, processo) Ex.: código exe da internet => a segurança baseada em código permite determinarmos que o programa não deve executar determinadas operações, como por exemplo, *ler ou gravar variáveis de ambiente*, *alterar o registro do Windows*, etc.

12. .net Framework - Características.

- Acesso **eficiente** a dados.
  - Sem **ODBC**.
  - Sem **BDE**.
  - O mecanismo de acesso é feito através de um ***Provider***. O ADO.net expõe uma interface padrão para todos os providers.
  - *A característica principal do provider é que ele se comunica "diretamente" com o banco de dados (exceto OLE DB e ODBC providers)*.
  - ***Dados desconectados***.
- Atributos.
  - **Obsolete**.
  - **Conditional**.
  - **Flags**.
  - Permite a definição de outros atributos.

13. .net Framework - Runtime.

- Compilação (**2 fases**).
  - Fase 1: Código fonte é transformado em um **assembly** (**IL**).
  - Fase 2: O compilador **JIT** (**Just In-Time compiler**) processa os byte codes da IL e **transforma em código nativo específico da plataforma** (*podendo sofrer otimizações*). O código é compilado conforme solicitação.
- **Managed Code** (Código Gerenciado).
  - **CLR** (**Common Language Runtime**).
    - ♦ Aplicações .net são executadas pelo CLR. Este ambiente, através da FCL, abstrai todo o sistema operacional.
  - **GC** (**Garbage Collector**).
    - ♦ ***Liberação automática*** de memória.

- Gerenciamento automático de **gerações** (0, 1 e 2).
- Fim do vazamento de memória (**Leak Memory**).

14. .net Framework - Linguagens.

- **Visual Basic .net**.
  - Linguagem ***adaptada*** para a plataforma.
  - ***Sofreu alterações significativas*** a ponto de ser considerada uma nova linguagem.
  - ***Herda os vícios da plataforma anterior***.
  - A linguagem **não é 100% orientada a objetos** (O compilador faz adaptações para o ambiente .net).
- **Delphi for .net**.
  - Linguagem ***adaptada*** para a plataforma.
  - ***Sofreu menos alterações em relação ao VB***.
  - ***Também herda os vícios da plataforma anterior***.
  - A linguagem **não é 100% orientada a objetos** (O compilador faz adaptações para o ambiente .net).

15. .net Framework - Linguagens.

- **C#.**
  - C# (C Sharp – **C Aprimorado**).
  - **Projetado especificamente para a plataforma .net.** (Nasceu para o .net).
  - *A arquitetura e as metodologias da linguagem C# refletem as metodologias subjacentes do .net*. Em muitos casos, as características da linguagem C#, de fato, dependem das características do .net, ou das classes-base do .net.
  - **Linguagem baseada na metodologia moderna do projeto orientado a objetos**.
  - Desenvolvido pela Microsoft: Equipe liderada por **Anders Hejlsberg** e **Scott Wiltamuth**.
  - **Fato!!! Grande parte das classes-base do .net foi escrita em C#.**
  - Linguagem orientada a objetos (***totalmente***).
  - Suporta sobrecarga de operadores... apesar do .net não suportar este recurso, o compilador C# traduz para uma chamada a método.
  - ***Conjunto coerente e bem definido de tipos básicos***.

16. .net Framework - Linguagens.

- **C#** (*continuação*).
  - Suporte para documentação XML.
  - Suporte para atributos.
  - Amplo acesso a FCL.
  - Fácil acesso a API do Windows (**raramente necessário**).
  - Possibilidade de uso de ponteiros e acesso direto a memória (**raramente necessário**).
  - ***Suporte a propriedades e eventos***.
  - **Bem mais fácil que C++.**
  - ***Simples/Moderno/Tipo Seguro (type-safe).***
  - Eficiência da linguagem C++ com a facilidade de ambientes RAD sem a complexidade e problemas de ponteiros, enfim... a junção de tudo o que há de melhor nesses últimos 20 anos.
  - ***O fato de C# ter sido projetado especificamente para a plataforma .net, significa que seu suporte ao .net framework é mais completo e também é oferecido em um contexto de uma sintaxe mais apropriada do que outras linguagens***.

17. .net Framework - Aplicações.

- Tipos de Aplicação.
  - **WebServices** (**XML**).
  - Windows Services.
  - **ASP.net** (**ThinClient**, requer apenas o browser no cliente): Ao contrário do ASP antigo, o código no lado servidor é compilado.
  - Console.
  - **Windows Forms** (**FatClient**, requer o .net Framework no cliente).

18. .net Framework - Conclusão.

- ***Plataforma***.
  - Provavelmente o que reinará nos próximos **20 anos**.
- ***Linguagem***.
  - **Escolha C#.**
- ***.net Compact Framework.***
  - O .net poderá ser portado para.
    - Pocket PC (já existe).
    - Celular (já existe).
    - Enfim... qualquer outro dispositivo, tornando sua aplicação funcional em outros ambientes.
- **Projeto Mono**.
  - .net Framework para outras plataformas (www.go-mono.com).
- **Reverse Engineer**.
  - ***Obfuscation***.

19. .net Framework.

- Dúvidas?

1. Apresentando C#.

2. C#

- Nascimento: **Dezembro de 1998**.
- Atributos: Linguagem de programação ***simples***, ***moderna***, ***type-safe*** e **totalmente orientada a objeto**.
- **type-safe**: Em C# é impossível ter problemas devido a variáveis que não foram inicializadas, ou acesso a arrays fora de seus limites ou ainda realizar casts inválidos.
- Ambiente: **.net Framework**.
- Versões da linguagem...
  - **1.0** (VS .net 2002 e 2003) – .net Framework 1.0 e 1.1.
    - Arquitetos responsáveis (Designers)...
      - ⇒ **Anders Hejlsberg**
      - ⇒ Scott Wiltamuth
      - ⇒ Peter Golde
      - ⇒ Peter Sollich
      - ⇒ Eric Gunnerson
  - **2.0** (VS .net 2005) – .net Framework 2.0.
    - Arquitetos responsáveis (Designers)...
      - ⇒ **Anders Hejlsberg**
      - ⇒ Peter Golde
      - ⇒ Peter Hallam
      - ⇒ Shon Katzenberger
      - ⇒ Todd Proebsting
      - ⇒ Anson Horton

3. Keywords...

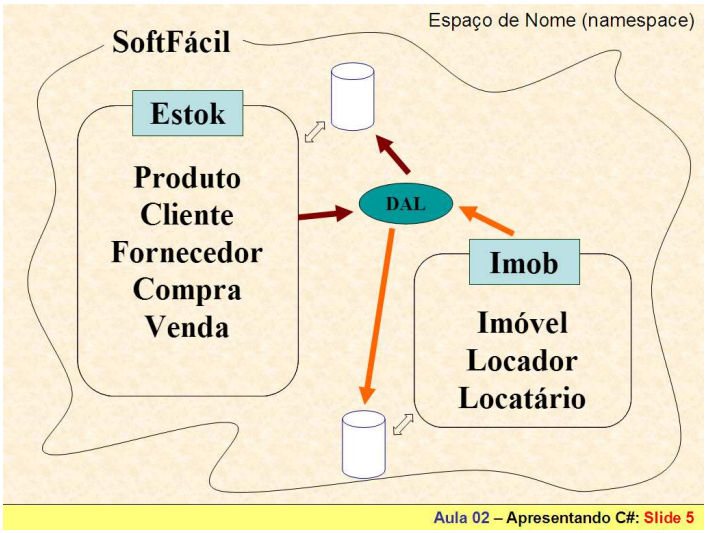
[ ] abstract	[ ] event	[ ] new	[ ] struct
[ ] as	[ ] explicit	[ ] null	[ ] switch
[ ] base	[ ] extern	[ ] object	[ ] this
[ ] bool	[ ] false	[ ] operator	[ ] throw
[ ] break	[ ] finally	[ ] out	[ ] true
[ ] byte	[ ] fixed	[ ] override	[ ] try
[ ] case	[ ] float	[ ] params	[ ] typeof
[ ] catch	[ ] for	[ ] private	[ ] uint
[ ] char	[ ] foreach	[ ] protected	[ ] ulong
[ ] checked	[ ] goto	[ ] public	[ ] unchecked
[ ] class	[ ] if	[ ] readonly	[ ] unsafe
[ ] const	[ ] implicit	[ ] ref	[ ] ushort
[ ] continue	[ ] in	[ ] return	[✓] using
[ ] decimal	[ ] int	[ ] sbyte	[ ] virtual
[ ] default	[ ] interface	[ ] sealed	[ ] volatile
[ ] delegate	[ ] internal	[ ] short	[ ] void
[ ] do	[ ] is	[ ] sizeof	[ ] while
[ ] double	[ ] lock	[ ] stackalloc	
[ ] else	[ ] long	[ ] static	
[ ] enum	[✓] namespace	[ ] string	

4. Hello World...

```
using System;
```

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

namespace Curso.CSharp.HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World.");
            Console.ReadLine();
        }
    }
}
```

5. <u>Espaço de Nome (namespace)...</u>



Exemplos...

Nova aplicação console... (Estok.exe)

```
// Produto.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.Estok
{
    class Produto
    {
    }
}

// Cliente.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.Estok
{
    class Cliente
    {
    }
```

```
}

// Fornecedor.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.Estok
{
    class Fornecedor
    {
    }
}

// CompraVenda.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.Estok
{
    class Compra
    {
    }

    class Venda
    {
    }
}

// DAL.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.Estok
{
    class Open
    {
        public static void OpenDB(string db)
        {
            Console.WriteLine(db + " aberto.");
        }
    }
}

// Program.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.Estok
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Estok.");
            Open.OpenDB("Estok.fdb");
            Console.ReadLine();
```

```
        }
    }
}
```

Nova aplicação console... (Imob.exe)

```
// Modulos.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.Imob
{
    class Imovel
    {
    }

    class Locador
    {
    }

    class Locatario
    {
    }
}

// DAL.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.Imob
{
    class Open
    {
        public static void OpenDB(string db)
        {
            Console.WriteLine(db + " aberto.");
        }
    }
}

// Program.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.Imob
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Imob.");
            Open.OpenDB("Imob.fdb");
            Console.ReadLine();
        }
    }
}
```

Criando um assembly (DAL.dll)...

```
// DAL.cs

using System;
using System.Collections.Generic;
using System.Text;

namespace SoftFacil.DataAccessLayer
{
    public class Open
    {
        public static void OpenDB(string db)
        {
            Console.WriteLine(db + " aberto.");
        }
    }
}
```

Utilizando o assembly em Estok e Imob... (Add Reference...)

```
SoftFacil.DataAccessLayer.Open.OpenDB("Estok.fdb");
SoftFacil.DataAccessLayer.Open.OpenDB("Imob.fdb");
```

ou...

```
using SoftFacil.DataAccessLayer;

Open.OpenDB("Estok.fdb");
Open.OpenDB("Imob.fdb");
```

6. C#...

- Que relação existe entre o nome da classe e seu espaço de nome (***fullname***) com o nome do arquivo que os contem?

7. De volta a Hello World...

8. DLL Hell...

- **Hell...**
  - **App1** acessa versão **1.0** de Lib.dll.
  - Instala-se **App2**... **App2** *sobrescreve* versão **1.0** de Lib.dll com versão **2.0** de Lib.dll.
  - **App2** *executa*.
  - **App1** ***não executa mais*** (Lib.dll incompatível).

- **Reinstala-se App1**.
- **App1** retorna versão **1.0** de Lib.dll.
- **App1** executa.
- ***App2 não executa mais***.
- Instala-se **App2**... **App2 executa**, mas **App1 não**.
- Instala-se **App1**... **App1 executa**, mas **App2 não**.
- Instala-se... enfim... ***DLL Hell***.

➢ **Windows 2000/XP...**

- Foi introduzido o recurso de **execução lado a lado**.
- Instalação da DLL no diretório da aplicação.
- **LoadLibrary()** da **Win32** foi re-escrita para **acessar primeiramente a pasta local**.

9. Assembly...

➢ Privado.
➢ Compartilhado *(**GAC**).*
➢ **Auto-descritivo (metadados).**
➢ **Não requer registro no Windows.**
➢ Manifesto controla versão e dependências.
➢ **Instalação de zero-impacto**.

Vamos criar um assembly compartilhado...

10. AssemblyInfo.cs...

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

[assembly: AssemblyTitle("DAL")]
[assembly: AssemblyDescription("Lib de acesso SoftFácil")]
[assembly: AssemblyConfiguration("Beta 2")]
[assembly: AssemblyCompany("SoftFácil")]
[assembly: AssemblyProduct("DAL")]
[assembly: AssemblyCopyright("Copyright © SoftFácil")]
[assembly: AssemblyTrademark("DAL marca registrada")]
[assembly: AssemblyCulture("")]

[assembly: ComVisible(false)]
[assembly: Guid("77fe6609-b9a2-4600-8ef2-a746cc333c4e")]

[assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

11. Assembly Compartilhado (shared)...

➢ **Criptografia...**

- Criptografia ***Simétrica***.
- Criptografia ***Assimétrica*** (Assinatura digital de um driver no Windows).
- **sn –k secret.snk**
- secret.snk (chaves *private* e *public*).

➢ ***DAL***.

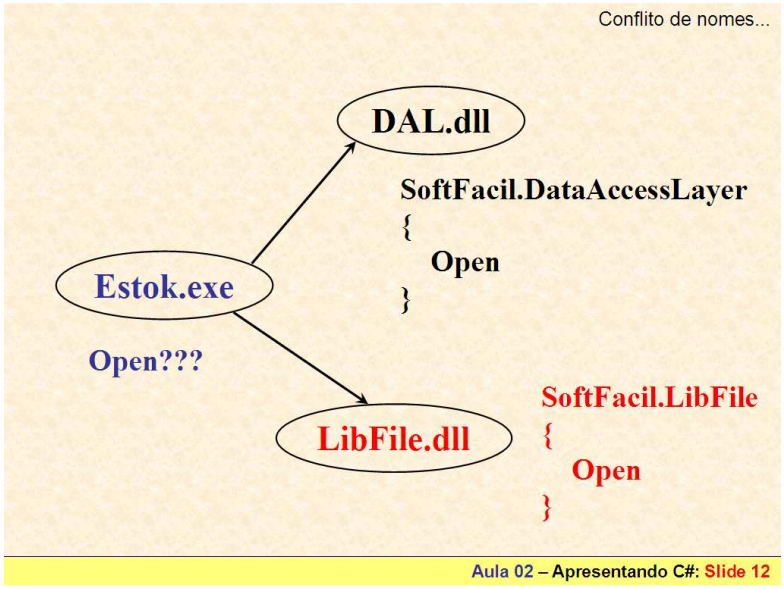
- Propriedades do Assembly... **Signing**.
- **[assembly: AssemblyVersion("1.0.*")]**
- ***Major**:* Versão Primária.
- ***Minor**:* Versão Secundária.
- ***Build**:* Número de Construção... Número de dias desde 1º de Janeiro de 2000.
- ***Revision**:* Número de Revisão... Número de segundos desde a meia noite da hora local.
- Instala-se o assembly no ***GAC***.
- **gacutil /i DAL.dll**

➢ *Consumindo o assembly*.

- Add Reference (DLL da pasta bin) – Copy Local é configurado como false.

- Aprimorando o assembly mas mantendo a versão principal e secundária.
- Fazendo uma mudança estrutural de 1.0 para 1.1.

12. Conflito de Nomes...



Novo assembly (LibFile.dll)...

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

namespace SoftFacil.LibFile
{
    public class Open
    {
        public static void OpenFile(string file)
        {
            Console.WriteLine("Arquivo {0} aberto.", file);
        }
    }
}
```

Utilizando o assembly LibFile.dll em Estok.exe (Add Reference)...

```
using System;
using System.Collections.Generic;
using System.Text;
using SoftFacil.DataAccessLayer;

namespace SoftFacil.Estok
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Estok.");
            Open.OpenDB("Estok.fdb");
            SoftFacil.LibFile.Open.OpenFile("Dados.txt");
            Console.ReadLine();
        }
    }
}
```

ou...

```
using System;
using System.Collections.Generic;
using System.Text;
using SoftFacil.DataAccessLayer;
using File = SoftFacil.LibFile.Open;

namespace SoftFacil.Estok
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Estok.");
            Open.OpenDB("Estok.fdb");
            File.OpenFile("Dados.txt");
            Console.ReadLine();
        }
    }
}
```

13. Unidade de Compilação (Estrutura geral)...

```
// Source File (.cs) – Compilation Unit

using ...

[GlobalAttributes ...]

namespace ...
{
    using ...
    ...
}

namespace ...
{
    using ...
    ...
}

Type1 ...
Type2 ...
```

14. namespace / using (segredos)...

```
namespace N1.N2
{
    class A // N1.N2.A
    {
    }

    class B // N1.N2.B
    {
    }
}
```

ou

```
namespace N1
{
    namespace N2
    {
```

```
        class A // N1.N2.A
        {
        }

        class B // N1.N2.B
        {
        }
    }
}
```

15. namespace / using (segredos)...

```
namespace N1.N2
{
    class A // N1.N2.A
    {
    }
}

namespace N1.N2
{
    class B // N1.N2.B
    {
    }
}
```

ou

```
namespace N1.N2
{
    class A // N1.N2.A
    {
    }

    class B // N1.N2.B
    {
    }
}
```

16. namespace / using (segredos)...

```
namespace N1.N2
{
    class A // N1.N2.A
    {
    }
}

namespace N3
{
    using A = N1.N2.A;

    class B : A // N3.B -> N1.N2.A
    {
    }
}
```

ou

```
namespace N1.N2
{
    class A // N1.N2.A
    {
    }
```

```
}

namespace N3
{
    using R = N1.N2;

    class B : R.A // N3.B -> N1.N2.A
    {
    }
}
```

17. namespace / using (segredos)...

```
namespace N1.N2
{
    class A // N1.N2.A
    {
    }
}

namespace N3
{
    using R1 = N1;
    using R2 = N1.N2;
    using R3 = N1.N2.A;

    class B
    {
        N1.N2.A obj1; // obj1 -> N1.N2.A
        R1.N2.A obj2; // obj2 -> N1.N2.A
        R2.A    obj3; // obj3 -> N1.N2.A
        R3      obj4; // obj4 -> N1.N2.A
    }
}
```

18. namespace / using (segredos)...

```
namespace N1.N2
{
    class A // N1.N2.A
    {
    }
}

namespace N3
{
    using N1.N2;

    class B : A // N3.B -> N1.N2.A
    {
    }
}
```

19. namespace / using (segredos)...

```
namespace N1
{
    class A // N1.A
    {
    }
}

namespace N2
{
```

```
    class A // N2.A
    {
    }
}

namespace N3
{
    using N1;
    using N2;

    class B : A // Erro (ambiguidade)
    {
    }
}
```

a solução é...

```
namespace N1
{
    class A // N1.A
    {
    }
}

namespace N2
{
    class A // N2.A
    {
    }
}

namespace N3
{
    using A = N1.A;
    using N2;

    class B : A
    {
    }
}
```

a solução também pode ser...

```
namespace N1
{
    class A // N1.A
    {
    }
}

namespace N2
{
    class A // N2.A
    {
    }
}

namespace N3
{
    using N2;

    class B : N1.A
    {
    }
```

```
}
```

20. C#...

- Dúvidas?

1. C# Fundamentos

2. Keywords...

[ ] abstract	[ ] event	[ ] new	[ ] struct
[ ] as	[ ] explicit	[ ] null	[ ] switch
[ ] base	[ ] extern	[✓] object	[ ] this
[✓] bool	[✓] false	[ ] operator	[ ] throw
[ ] break	[ ] finally	[ ] out	[✓] true
[✓] byte	[ ] fixed	[ ] override	[ ] try
[ ] case	[✓] float	[ ] params	[ ] typeof
[ ] catch	[ ] for	[ ] private	[✓] uint
[✓] char	[ ] foreach	[ ] protected	[✓] ulong
[✓] checked	[ ] goto	[ ] public	[✓] unchecked
[ ] class	[ ] if	[ ] readonly	[ ] unsafe
[ ] const	[ ] implicit	[ ] ref	[✓] ushort
[ ] continue	[ ] in	[ ] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[ ] virtual
[ ] default	[ ] interface	[ ] sealed	[ ] volatile
[ ] delegate	[ ] internal	[✓] short	[ ] void
[ ] do	[ ] is	[ ] sizeof	[ ] while
[✓] double	[ ] lock	[ ] stackalloc	
[ ] else	[✓] long	[ ] static	
[ ] enum	[✓] namespace	[✓] string	

3. Comentários

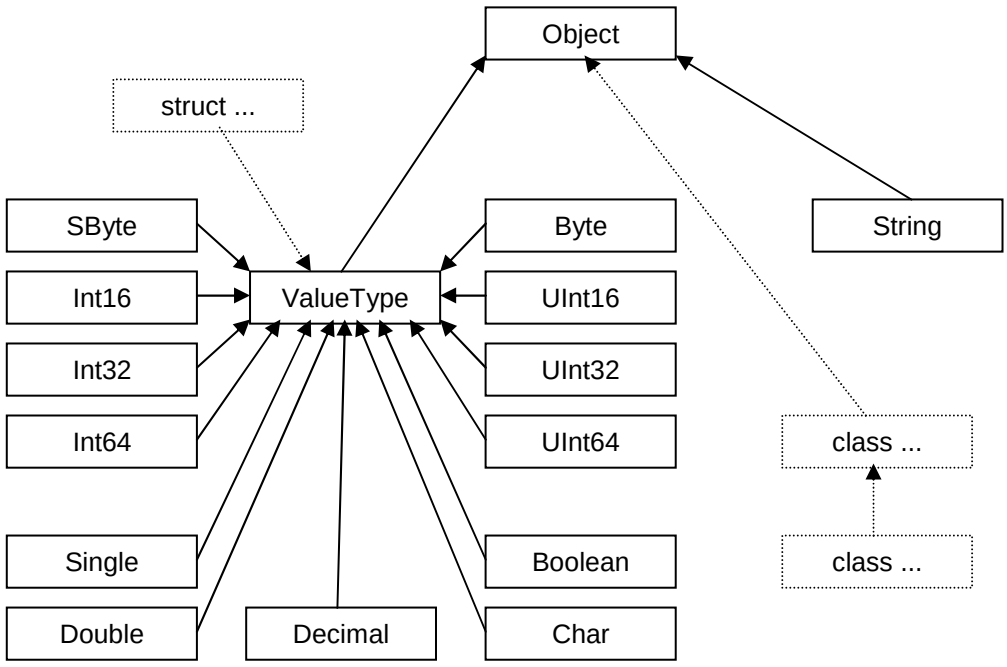
- Comentários de uma única linha (//).
- Comentários de mais de uma linha (/* ... */).

```
/*
 * Hello World
 */

using System;
using System.Collections.Generic;
using System.Text;

namespace Curso.CSharp.HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World.");
            Console.ReadLine(); // Aguarda pressionamento da tecla Enter
        }
    }
}
```

4. Tipos de dados pré-definidos

C# type	CTS Common Type System	Intervalo / Descrição	bits
Tipos Valor (Stack/Pilha)			
Inteiro			
sbyte	System.SByte	-128 ... 127	8
short	System.Int16	-32,768 ... 32,767	16
int	System.Int32	-2,147,483,648 ... 2,147,483,647	32
long L	System.Int64	-9,223,372,036,854,775,808 ... 9,223,372,036,854,775,807	64
byte	System.Byte	0 ... 255	8
ushort	System.UInt16	0 ... 65,535	16
uint U	System.UInt32	0 ... 4,294,967,295	32
ulong UL	System.UInt64	0 ... 18,446,744,073,709,551,615	64
Ponto Flutuante			
float F	System.Single	7 dígitos de precisão = $\pm 1.5 \times 10^{-45}$ ... $\pm 3.4 \times 10^{38}$	32
double D	System.Double	15 dígitos de precisão = $\pm 5 \times 10^{-324}$ ... $\pm 1.7 \times 10^{308}$	64
Decimal (valores monetários)			
decimal M	System.Decimal	28 dígitos de precisão = $\pm 1.0 \times 10^{-28}$ ... 7.9×10^{28}	128
Booleano			
bool	System.Boolean	true ou false (não aceita 0 ou 1)	8
Caractere			
char	System.Char	Um único caractere (Unicode 16 bits) '\u0000' a '\uFFFF' ASCII é um subconjunto do Unicode	16
Tipos Referência (Heap)			
object	System.Object	Tudo deriva daqui (raiz da hierarquia de tipos)	
string	System.String	String de caracteres unicode	

5. Hierarquia




```
int x = 1; // Int32 x = 1;
Console.WriteLine(x.ToString());
```

Gerenciamento de memória: stack / heap.

6. Declaração e inicialização de variáveis

tipo identificador;

tipo identif1, identif2, identif3;
tipo identif = valor;
tipo identif1 = valor1, identif2 = valor2, identif3 = valor3;

tipo identif1, identif2, identif3;
identif1 = identif2 = identif3 = valor;

Exemplos...

```
short v1;
int w1, w2, w3;
long v2 = 2;
byte x = 1, y = 2, z = 3;

int a, b, c;
a = b = c = 0;
```

7. Identificadores...

- PascalCase / camelCase.
- Case-sensitive.
- Deve começar por **letra** ou sublinhado (**underscore**).
- Pode conter **letra**, **sublinhado** ou **número**.
- ***Acentos são permitidos***.
- Palavras chaves podem ser usadas como identificadores através do uso de **verbatim** (@).
  - Muito útil caso uma classe chamada "**bool**" tenha sido criada, por exemplo, em *Visual Basic*, e a estejamos consumindo em C#.
- Tamanho!!! **qual o limite**?

8. Declaração e inicialização de variáveis (exemplos)...

```
sbyte v1;
int identif1, identif2, identif3;
long v2 = 2;
float x = 1.0F, y = 2, z = 3;

int a, b, c, d;
a = b = c = 0;
d = 1;

decimal salario = 9900.50M;
bool @using = true;

char letra = 'A';
string msg = "Hello";

byte caractere1 = 255;
byte caractere2 = 0xFF;

char letraA1 = 'A';
char letraA2 = '\x0041';
char letraA3 = '\u0041';
char letraA4 = (char)65;

ushort codigo = (ushort)letraA1;
```

9. Sequências de escape (\)...

- \' Aspas Simples
- \" Aspas Dupla
- \\ Barra Invertida
- \0 Nulo (\u0000 = 0)
- \a Alerta (\u0007 = 7)

- \b Backspace (\u0008 = 8)
- \f Form Feed (\u000C = 12)
- \n New Line (\u000A = 10)
- \r Carriage Return (\u000D = 13)
- \t Tabulação Horizontal (\u0009 = 9)
- \v Tabulação Vertical (\u000B = 11)
- Caractere Unicode...
  - \x0000 ... \xFFFF
  - \u0000 ... \uFFFF (ou \U)

10. Exemplos...

```
char aspasSimples = '\'';
char aspasDupla = '"';
string msg1 = "Great \"Power\"."; // Great "Power".

string msg = "C#\nLinguagem .net";

string file1 = "C:\\Temp\\Dados.txt";
string file2 = "C:/Temp/Dados.txt";
string file3 = @"C:\Temp\Dados.txt"; // string verbatim

string sql = @"
    SELECT Id, Nome, Fone
    FROM Cliente
    WHERE UF = ""SP""
";
```

11. Strings (imutáveis)...

```
// Tipo Valor (Stack)

int x = 1;
int y = 2;
int z = x; // z = 1
z = 3; // x = 1, y = 2, z = 3

// Tipo Referência (Heap)

string s1 = "texto 1";
string s2 = "texto 2";
string s3 = s1; // ponteiro para s1
s3 = "texto 3"; // s3 = "texto 3"
s3 = s3 + " três"; // s3 = "texto 3 três"
// "texto 3" continua a existir
// mas está disponível para o GC
```

12. Operadores (Precedência)...

- 01º... () . [] x++ x-- new← typeof sizeof← checked unchecked()
- 02º... + - ! ~ ++x --x (tipo) (←)
- 03º... * / %
- 04º... + -
- 05º... << >>
- 06º... < > <= >= is as
- 07º... == !=
- 08º... &
- 09º... ^
- 10º... |
- 11º... &&
- 12º... ||
- 13º...?: (←)
- 14º... = += -= *= /= %= &= |= ^= <<= >>= (←)

```
w = x + y * 2;
if (x + 1 == 4 && y > 2) ...
a = b = c = 5;
```

13. Operadores (Precedência)...

- 1º...
  - Parênteses: ()
  - Acesso a membros: .
  - Indexação: []
  - Pós-incremento: ++
  - Pós-decremento: --
  - Criação de objeto: new
  - Informação de tipo: typeof
  - Tamanho em bytes na pilha: sizeof
  - Controle de overflow: checked(...), unchecked(...)
  - *Statements (não são operadores)*...
    - checked { ... unchecked { ... } ... }
    - unchecked { ... }

```
byte a = 1;
int b;
b = (a + 5) * 2;
Console.WriteLine(b); // 12
b++;
Console.WriteLine(b); // 13
b = 2147483647;
b = b + 1;
Console.WriteLine(b); // -2147483648
b = 2147483647;
b = checked(b + 1); // overflow
// ou
checked
{
    b = b + 1;
}
```

14. Operadores (Precedência)...

- 2º (Unários)...
  - Sinal: +, -
  - Negação lógica (Not): !
  - Negação binária: ~
  - Pré-incremento: ++
  - Pré-decremento: --
  - Casting: (tipo)

```
sbyte a = -1;
bool b = true;
bool c = !b; // c = false
byte d = 0;
d = (byte)~d; // d = 255
int e, f;
e = f = 5;
int g = e++; // g = 5, e = 6
int h = ++f; // h = 6, f = 6
```

15. Operadores (Precedência)...

- 3º...
  - Multiplicação: *

- Divisão: /
- Módulo: %

➢ 4º...

- Adição/Concatenação: +
- Subtração: -

➢ 5º...

- Deslocamento de bit a esquerda: <<
- Deslocamento de bit a direita: >>

```
int a = 5 * 2;  // a = 10
int b = 11 / 4; // b = 2 (quociente inteiro)
int c = 11 % 4; // c = 3 (resto inteiro)
int d = 5 + 1;  // d = 6
int e = 4 - 2;  // e = 2
string f = "C# " + "Language."; // f = "C# Language."
int g1 = 5 << 1; // 5 * 2
int g2 = 5 << 2; // 5 * 4
int g3 = 5 << 3; // 5 * 8
int h1 = 5 >> 1; // 5 / 2
int h2 = 5 >> 2; // 5 / 4
int h3 = 5 >> 3; // 5 / 8
double i = 5   / 2; // i = 2
double j = 5.0 / 2; // j = 2.5
double k = 5D  / 2; // k = 2.5
```

16. Operadores (Precedência)...

➢ 6º (Relacionais)...

- Menor que: <
- Maior que: >
- Menor ou igua a: <=
- Maior ou igual a: >=
- Informação de tipo: is, as

➢ 7º (Comparação)...

- Igual a: ==
- Diferente de: !=

```
char sexo = 'M';
if (sexo == 'M')
{
    Console.WriteLine("Homem");
}
else
{
    Console.WriteLine("Mulher");
}
```

17. Operadores (Precedência)...

➢ 8º... And bit-a-bit: &
➢ 9º... XOr bit-a-bit: ^
➢ 10º... Or bit-a-bit: |
➢ 11º... And booleano: &&
➢ 12º... Or booleano: ||

```
char sexo = 'F';
int idade = 20;
if (sexo == 'F' && idade > 18)
{
    Console.WriteLine("Mulher adulta");
}
```

18. Operadores (Precedência)...

- 13º (Ternário)... (associatividade: da direita para a esquerda)
  - Condicional: ?:
- 14º (Atribuição)... (associatividade: da direita para a esquerda)
  - =
  - +=
  - -=
  - *=
  - /=
  - %=
  - &=
  - |=
  - ^=
  - <<=
  - >>=

```
char sexo = 'M';
Console.WriteLine(sexo == 'M' ? "Homem" : "Mulher");

int tot = 0;
tot = tot + 1;
// ou
tot++;
// ou
++tot;
// ou
tot += 1;
```

19. Exponenciação...

- Não existe operador para exponenciação em C#.
- Math.Pow().

```
double r = Math.Pow(5, 2); // r = 5 ao quadrado = 25
```

20. Conversões...

- ***Implícitas***...
  - De sbyte para short, int, long, float, double, decimal.
  - De byte para short, ushort, int, uint, long, ulong, float, double, decimal.
  - De short para int, long, float, double, decimal.
  - De ushort para int, uint, long, ulong, float, double, decimal.
  - De int para long, float, double, decimal.
  - De uint para long, ulong, float, double, decimal.
  - De long, ulong para float, double, decimal (**repare no float como conversão implícita, o resultado é menos preciso, mas a magnitude do número é preservada**).
  - De float para double.
  - **De char para ushort, int, uint, long, ulong, float, duble, decimal**.
  - ***Via de regra... sempre de um tipo menor para um tipo maior***.
- ***Explícitas***...
  - De modo análogo, toda conversão de um tipo maior para um menor deverá ser feita explicitamente através de ***casting***.

```
short a = 1000;
byte b1 = (byte)a; // não causa overflow
byte b2 = checked((byte)a); // causa overflow
double preco = 8.9;
int valor1 = (int)preco; // valor1 = 8
int valor2 = (int)(preco + 0.5); // Arredonda para o inteiro mais próximo
char c = (char)65; // caractere 'A'
int codigoCaractereA = 'A'; // = 65 (conversão implícita)
```

21. Conversões...

- ***Entre números e strings...***
  - De número para string...

```
int i = 10;
string s = i.ToString();
```

  - De string para número...

```
string s = "10";
int i = int.Parse(s);
```

22. Ponto Flutuante...

1.0/0.0 = +Infinito
-1.0/0.0 = -Infinito
0.0/0.0 = NaN (Not-a-Number)

```
double x =  1.0 / 0; // +Infinito
double y = -1.0 / 0; // -Infinito
double z =  0.0 / 0; // NaN (Not-a-Number)

Console.WriteLine(x);
Console.WriteLine(y);
Console.WriteLine(z);
```

23. System.Console (noções básicas)

- Console.Write(...);
- Console.WriteLine("{0} mais {1} é igual a {2}.", i, j, i + j);
- {argumento[,largura[:formato]]}
- +largura : alinha a direita
- -largura : alinha a esquerda
- Formato...
  - Cn ou cn: monetário com n casas decimais.
  - Dn ou dn: preenche um inteiro com n zeros.
  - E ou e: formato exponencial.
  - Fn ou fn: número de ponto flutuante com n casas decimais.
  - G ou g: usa a formatação E ou F dependendo do que for mais compacto.
  - N ou n: formato numérico estilo 999.999.999,99 (Brasil).
  - P ou p: formato percentual.
  - Xn ou xn: formato hexadecimal de n dígitos.

24. Main

- Ponto de entrada (startup) de qualquer executável em C#.
- Possíveis assinaturas...
  - `static void Main(string[] args)`
  - `static void Main()`
  - `static int Main(string[] args)`
  - `static int Main()`

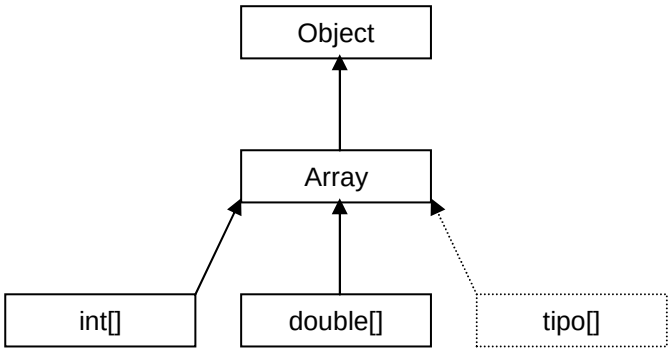
25. Boxing/Unboxing

- ***Boxing*** é a transformação de um ***tipo valor*** em um ***tipo referência*** (**da pilha para o heap**).
- ***Unboxing*** é a transformação de um ***tipo valor alocado no heap*** para um ***tipo valor alocado na pilha***.

```
int i = 10;
object obj = i; // boxing
int j = (int)obj; // unboxing
```

26. Arrays

- Estrutura de dados que contem elementos.
- Um objeto da classe base ***System.Array*** é instanciado **quando inicializamos um array**.
- Não há risco de vazamento de memória (utilização de um elemento fora dos limites do array), pois os arrays em C# são **type-safe**.
- Tipos...
  - Regulares...
    - Vetoriais (unidimensionais)...
    - Matrizes (multidimensionais)...
  - Irregulares...
    - Arrays de arrays.




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

variavel[0], variavel[1], ... variavel[n – 1]
```

27. Vetores

```
int[] n1; // um array “não inicializado” de inteiros (n1 = null)

// n2, n3, n4 e n5 são arrays de 5 elementos (0..4)
// contendo os valores 10, 20, 30, 40 e 50
int[] n2 = { 10, 20, 30, 40, 50 };
int[] n3 = new int[] { 10, 20, 30, 40, 50 };
int[] n4 = new int[5] { 10, 20, 30, 40, 50 };
int[] n5 = new int[5];
n5[0] = 10;
n5[1] = 20;
n5[2] = 30;
n5[3] = 40;
n5[4] = 50;

// n6 é um array de inteiros contendo 10 elementos (0..9)
// inicializados com o valor zero
int[] n6 = new int[10];

      string[] languages = { "C#", "Delphi", "VB" };
ou... string[] languages = new string[] { "C#", "Delphi", "VB" };
ou... string[] languages = new string[3] { "C#", "Delphi", "VB" };
```

28. Matrizes

```
int[,] numeros = new int[3, 2] {
    { 1, 2 },
    { 3, 4 },
    { 5, 6 }
};
```

ou...

```
int[,] numeros = new int[3, 2];
numeros[0, 0] = 1;
numeros[0, 1] = 2;
numeros[1, 0] = 3;
numeros[1, 1] = 4;
numeros[2, 0] = 5;
numeros[2, 1] = 6;
```

29. Arrays irregulares

```
double[][] numeros = new double[3][];
numeros[0] = new double[2] { 1.0, 2.0 };
numeros[1] = new double[4] { 3.0, 4.0, 5D, 6D };
numeros[2] = new double[1] { 7D };
```

30. C#...

- Dúvidas?

1. Estruturas de Controle

2. Keywords...

[ ] abstract	[ ] event	[ ] new	[ ] struct
[ ] as	[ ] explicit	[ ] null	[✓] switch
[ ] base	[ ] extern	[✓] object	[ ] this
[✓] bool	[✓] false	[ ] operator	[ ] throw
[✓] break	[ ] finally	[ ] out	[✓] true
[✓] byte	[ ] fixed	[ ] override	[ ] try
[✓] case	[✓] float	[ ] params	[ ] typeof
[ ] catch	[✓] for	[ ] private	[✓] uint
[✓] char	[✓] foreach	[ ] protected	[✓] ulong
[✓] checked	[✓] goto	[ ] public	[✓] unchecked
[ ] class	[✓] if	[ ] readonly	[ ] unsafe
[ ] const	[ ] implicit	[ ] ref	[✓] ushort
[✓] continue	[✓] in	[ ] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[ ] virtual
[✓] default	[ ] interface	[ ] sealed	[ ] volatile
[ ] delegate	[ ] internal	[✓] short	[ ] void
[✓] do	[ ] is	[ ] sizeof	[✓] while
[✓] double	[ ] lock	[ ] stackalloc	
[✓] else	[✓] long	[ ] static	
[ ] enum	[✓] namespace	[✓] string	

3. Blocos de Código

```
{
  declarações...
  instruções...
  {
    declarações...
    instruções...
  }
}
```

4. if/else

```
if (condição)
    instrução;

if (condição)
    instrução;
else
    instrução;

if (condição)
{
    instruções...
}

if (condição)
{
    instruções...
```

```
}
else
{
    instruções...
}
```

5. if/else

Exemplo 1

```
if (idade >= 18 && sexo == 'M')
{
    Console.WriteLine("Homem maior de idade.");
}
```

Exemplo 2

```
if (cargo == "Presidente")
{
    // Presidente
}
else if (cargo == "Analista")
{
    // Analista
}
else if (cargo == "Telefonista")
{
    // Telefonista
}
else
{
    // Outros
}
```

6. goto

```
label:
  instrução;
  instrução;
  ...
  goto label;
```

Exemplo

```
{
inicio:
    Console.Write("C# ");
    goto inicio;
}
```

7. switch

```
switch (variável)
{
    case constante1:
        ...
        break;
    case constante2:
        ...
        goto case constante;
    case constante3:
    case constante4:
        ...
```

```
        break;
    default:
        ...
        break;
}
```

8. switch

```
string pais = "Brasil";
string lingua = string.Empty;
switch (pais)
{
    case "Brasil":
    case "Portugal":
        lingua = "Português";
        break;
    case "EUA":
        lingua = "Inglês";
        break;
    default:
        Console.WriteLine("País não cadastrado.");
        goto case "EUA";
}
```

9. for

```
for (inicializador; condição; iteração)
{
    ...
    break; // aborta o laço
    continue; // executa a próxima iteração, depois avalia a condição
}
```

Exemplo

```
for (int i = 0; i < 10; i++)
{
    if (i == 6)
    {
        continue;
    }
    Console.WriteLine(i);
}
```

10. while

```
while (condição)
{
    ...
    break; // aborta o laço
    continue; // avalia a condição e continua ou não o laço
}
```

Exemplo

```
int i = 0;
while (i < 10)
{
    Console.WriteLine(i);
    i++;
}
```

11. do ... while

```
do
{
    ...
    break; // aborta o laço
    continue; // avalia a condição e continua ou não o laço
} while (condição);
```

Exemplo

```
int i = 0;
do
{
    Console.WriteLine(i);
    i++;
} while (i < 10);
```

12. foreach

```
foreach (tipo variável in collection)
{
    ... // variável é read-only
    break; // aborta o laço
    continue; // processa o próximo item na coleção
}
```

Exemplo

```
int[] numeros = { 1, 2, 3, 4, 5 };
foreach (int numero in numeros)
{
    Console.WriteLine(numero);
    // numero é read-only
}
```

ou...

```
for (int i = 0; i < numeros.Length; i++)
{
    Console.WriteLine(numeros[i]);
    // Neste caso, numeros[i] pode ser alterado
}
```

13. Bhaskara (Equação do 2º Grau)

$$ax^2 + bx + c = 0 \quad (a \neq 0)$$

$$\Delta = b^2 - 4ac$$

$\Delta < 0 \quad \rightarrow$ Não existe raiz real

$$\Delta = 0 \quad \rightarrow \quad x = \frac{-b}{2a}$$

$$\Delta > 0 \quad \rightarrow \quad x_1 = \frac{-b + \sqrt{\Delta}}{2a} \qquad x_2 = \frac{-b - \sqrt{\Delta}}{2a}$$

14. Bhaskara

```
bool finalizaPrograma = false;
do
```

```
{
    double a, b, c, delta;
    do
    {
        Console.Write("a=");
        a = double.Parse(Console.ReadLine());
    } while (a == 0);
    Console.Write("b="); b = double.Parse(Console.ReadLine());
    Console.Write("c="); c = double.Parse(Console.ReadLine());
    delta = b * b - 4 * a * c;
    if (delta < 0)
    {
        Console.WriteLine("Não existe raiz real.");
    }
    else
    {
        if (delta == 0)
        {
            double x = -b / (2 * a);
            Console.WriteLine("x = {0}", x);
        }
        else
        {
            double x1, x2;
            x1 = (-b + Math.Sqrt(delta)) / (2 * a);
            x2 = (-b - Math.Sqrt(delta)) / (2 * a);
            Console.WriteLine("x1 = {0}", x1);
            Console.WriteLine("x2 = {0}", x2);
        }
    }
    Console.Write("Finalizar S/N?");
    finalizaPrograma = Console.ReadLine() == "S";
} while (!finalizaPrograma);
```

15. C#

- Dúvidas?

1. Programação Orientada a Objetos (Object Oriented Programming – OOP)

2. Keywords...

[✓] abstract	[ ] event	[✓] new	[ ] struct
[✓] as	[✓] explicit	[✓] null	[✓] switch
[✓] base	[✓] extern	[✓] object	[✓] this
[✓] bool	[✓] false	[✓] operator	[ ] throw
[✓] break	[ ] finally	[✓] out	[✓] true
[✓] byte	[ ] fixed	[✓] override	[ ] try
[✓] case	[✓] float	[✓] params	[✓] typeof
[ ] catch	[✓] for	[✓] private	[✓] uint
[✓] char	[✓] foreach	[✓] protected	[✓] ulong
[✓] checked	[✓] goto	[✓] public	[✓] unchecked
[✓] class	[✓] if	[✓] readonly	[ ] unsafe
[✓] const	[✓] implicit	[✓] ref	[✓] ushort
[✓] continue	[✓] in	[✓] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[✓] virtual
[✓] default	[ ] interface	[✓] sealed	[ ] volatile
[ ] delegate	[✓] internal	[✓] short	[✓] void
[✓] do	[✓] is	[ ] sizeof	[✓] while
[✓] double	[ ] lock	[ ] stackalloc	
[✓] else	[✓] long	[✓] static	
[ ] enum	[✓] namespace	[✓] string	

3. Introdução

- Linguagem de programação...
  - Procedimental (Orientada a Ações). A unidade de programação é a função.
    - Visual Basic
    - Pascal
    - Cobol
    - Clipper
    - C, C++
  - Orientada a Objetos. A unidade de programação é a classe.
    - C#
  - Mista. Suporta programação por classe e procedimental.
    - Delphi
    - Delphi for .net
    - Visual Basic .net

4. OOP

- Não é suficiente aprender apenas a *sintaxe básica* de uma linguagem de programação...
  - Declaração de variáveis.
  - Controle de fluxo.
  - Escopo, etc.
- É necessário e fundamental construir ***código de qualidade***...
  - Um profundo conhecimento dos ***princípios e metodologias*** por trás da linguagem irá garantir a construção de um **código moderno e mais eficiente**.
  - Na **OOP**...
    - O principal fundamento é a ***herança***.

⇒ Herança de Implementação (classe base).
⇒ Herança de Interface (uma ou mais interfaces).

5. OOP e o mundo real

- Considere um objeto do mundo real... (***o rádio de seu carro***)
  - Você sabe *trocar as estações* de rádio.
  - Você sabe como *regular o volume*.
  - Você sabe como *reproduzir um CD de áudio*.
  - Mas...
    - Você sabe como ele funciona internamente?
    - Você sabe o número de rotações necessárias para a leitura do CD?
    - Você conhece os fundamentos de Hertz na sintonização da estação?
- O rádio... na **OOP** é um **objeto**...
  - Os controles de regulagem de estação, volume, e CD são os mecanismos com os quais o usuário tem acesso direto... dizemos na OOP que trata-se da **Interface**.
  - *O funcionamento interno, na realidade, não interessa a pessoa que utiliza o dispositivo*. Neste caso, na OOP, dizemos que o funcionamento interno esta **encapsulado**.

6. OOP e o mundo real

- Se trocarmos o rádio por um ***modelo mais novo***...
  - O novo modelo, muito provavelmente, foi construído pelo fabricante levando-se em consideração as *funcionalidades já prontas no modelo anterior* (diz se que ***o novo modelo herdou as caracteristicas do anterior***, mas é claro, acrescentando novos recursos).
  - Seu *funcionamento interno*, provavelmente *será totalmente diferente*.
  - Mas tudo bem... isso não interessa para o usuário do rádio... pois *a interface basicamente se manteve a mesma*...
  - Desta forma, o usuário, sem maiores problemas, conseguirá utilizar o novo rádio (**o novo objeto**).
- **OOP** na ***Engenharia de Software***...
  - Este mesmo princípio deve ser adotado na construção de software em C#.
    - Encapsulamento.
    - Interface.
    - Herança.

7. Classes

- *Modelos* com base nos quais podemos *criar objetos*.
- Objetos instanciados de uma determinada classe, passam a ter as características definidas na classe.
- Membros da classe...
  - **Dados (membros de armazenamento)**...
    - Constantes.
    - Campos.
    - Eventos.
  - **Funções (membros que contém código)**...
    - Construtores.
    - Destrutor.
    - Propriedades.
    - Métodos.
    - Indexadores.
    - Operadores.
  - Outros tipos (**nested types**)...
    - Classes, estruturas, etc.

8. Instanciando um objeto

```
Tipo identificador = new Tipo(...);

// Declara um identificador do tipo DateTime,
```

```
// cria uma instância de um objeto DateTime na pilha
// e o inicializa com 25 de Dezembro de 2005.
DateTime natal = new DateTime(2005, 12, 25);
// Declara três identificadores do tipo int (n1, n2 e n3),
// cria três instâncias de objetos int na pilha
// e os inicializa com o valor 0 (zero).
Int32 n1 = new Int32();
int n2 = new int();
int n3 = 0;
// Declara um identificador do tipo int,
// mas nenhum objeto é instanciado...
// n4 não está inicializado.
int n4;
// Declara um identificador do tipo Horario,
// cria uma instância de um objeto no heap
// e o inicializa com 9 horas 10 minutos e 20 segundos.
Horario horario = new Horario(9, 10, 20);
```

9. Assembly: Tempo.dll

```
namespace Curso.CSharp.Tempo
{
    class Horario
    {
        byte hora, minuto, segundo;

        Horario(byte h, byte m, byte s)
        {
            hora = h;
            minuto = m;
            segundo = s;
        }
    }
}
```

10. Modificadores de Acesso

```
namespace ...
{
    internal
    public      class Horario
    {
        ...
        private
        public
        internal            Horario(...)
        protected
        protected internal

        {
            ...
        }
    }
}
```

11. Modificadores de Acesso

- **private**: o recurso é acessível *somente dentro do tipo* onde foi declarado... o recurso faz parte do funcionamento interno (encapsulamento).
- **public**: o recurso é acessível *dentro e fora* do assembly.

- **internal**: o recurso é acessível *somente dentro do próprio assembly*... trata-se de uma espécie de *"public"* limitado ao próprio assembly.
- **protected**: o recurso é acessível somente *dentro do tipo onde foi declarado ou em tipos derivados deste* (herança).
- **protected internal**: "*protected or internal*"... indicando que o recurso é visto como um "public" dentro do próprio assembly e, fora dele, somente tipos derivados tem acesso.

12. Implementação ideal

```
public class Horario : object
{
    byte hora, minuto, segundo;

    public Horario(byte hora, byte minuto, byte segundo) : base()
    {
        this.hora = hora;
        this.minuto = minuto;
        this.segundo = segundo;
    }
}
```

13. Mantendo os campos consistentes

```
public class Horario : object
{
    byte hora, minuto, segundo;

    public Horario(byte hora, byte minuto, byte segundo) : base()
    {
        this.hora = hora >= 0 && hora <= 23 ? hora : (byte)0;
        this.minuto = minuto >= 0 && minuto <= 59 ? minuto : (byte)0;
        this.segundo = segundo >= 0 && segundo <= 59 ? segundo : (byte)0;
    }
}
```

14. Propriedades

- A idéia por trás das propriedades é que um ***método*** ou ***grupo de métodos*** seja revestido de tal forma que *pareça um campo*, no que diz respeito a qualquer código cliente.
- As propriedades podem ser...
  - read-write...
    - ♦ Acessores...
      - ⇒ get
      - ⇒ set
  - read-only...
    - ♦ Acessor...
      - ⇒ get (somente)
  - write-only (aconselhável a criação de um método **Set...()**, neste caso)...
    - ♦ Acessor
      - ⇒ set (somente)

15. Propriedades

```
public class Horario : object
{

    ...

    public byte Hora
    {
        get
        {
            return hora;
```

```
        }
        set
        {
            hora = value;
        }
    }
}
```

16. Propriedades (mantendo consistência)

```
public class Horario : object
{

    ...

    public byte Hora
    {
        get
        {
            return hora;
        }
        set
        {
            if (value >= 0 && value <= 23)
            {
                hora = value;
            }
        }
    }
}
```

17. Re-utilização (Boa Engenharia de Software)

```
public class Horario : object
{
    byte hora, minuto, segundo;

    public Horario(byte hora, byte minuto, byte segundo) : base()
    {
        Hora = hora;
        Minuto = minuto;
        Segundo = segundo;
    }

    public byte Hora
    {
        get
        {
            return hora;
        }
        set
        {
            if (value >= 0 && value <= 23)
            {
                hora = value;
            }
        }
    }

    public byte Minuto
    {
        get
        {
            return minuto;
```

```
        }
        set
        {
            if (value >= 0 && value <= 59)
            {
                minuto = value;
            }
        }
    }

    public byte Segundo
    {
        get
        {
            return segundo;
        }
        set
        {
            if (value >= 0 && value <= 59)
            {
                segundo = value;
            }
        }
    }
}
```

18. Sobrecarga (overload) de construtores

```
public Horario(byte hora, byte minuto, byte segundo) : base()
{
    Hora = hora;
    Minuto = minuto;
    Segundo = segundo;
}

public Horario(byte hora, byte minuto) : this(hora, minuto, 0)
{
}

public Horario(byte hora) : this(hora, 0, 0)
{
}

public Horario() : this(0, 0, 0)
{
}
```

19. Métodos

```
// Um método que não retorna nenhum valor,
// deve ser declarado como void.
public void AddHoras(int horas)
{
    Hora = (byte)((Hora + horas) % 24);
}

public string ToHora12()
{
    return string.Format("{0:d2}:{1:d2}:{2:d2}",
        Hora == 0 || Hora == 12 ? 12 : Hora % 12,
        Minuto,
        Segundo
        ) + (Hora < 12 ? " AM" : " PM");
}
```

```
public string ToHora24()
{
    return string.Format("{0:d2}:{1:d2}:{2:d2}",
        Hora,
        Minuto,
        Segundo
        );
}
```

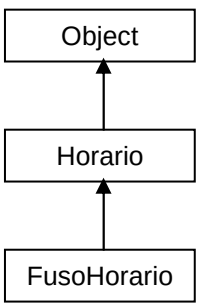
20. Inicialização e escopo de variáveis...

- O compilador C# dá ***enfase a segurança*** quando o assunto é inicialização de variáveis, ou seja, ***toda variável deve ser inicializada*** antes de ser trabalhada.
- Variáveis que são *campos de uma classe ou estrutura são zeradas por padrão* tão logo uma instância da classe ou estrutura seja criada.
- *Variáveis locais a um método deverão ser explicitamente inicializadas* antes de qualquer instrução na qual elas apareçam.
- Escopo...
  - Região de código onde a variável de fato existe.
  - Uma **variável de classe** (**campo**) existe dentro da classe.
  - Uma **variável local** existe dentro do bloco no qual foi declarada.
  - Uma variável declarada num laço **for** tem escopo somente dentro deste laço.

21. Herança

```
public class FusoHorario : Horario
{
    string pais;

    public FusoHorario(string pais, byte hora, byte minuto, byte segundo)
           : base(hora, minuto, segundo)
    {
        this.pais = pais;
    }

    new public string ToHora12()
    {
        return pais + ", " + base.ToHora12();
    }

    new public string ToHora24()
    {
        return pais + ", " + base.ToHora24();
    }
}
```

22. Relacionamento (é um) is




```
Horario horario = new Horario(9, 10, 2);
FusoHorario fusoHorario = new FusoHorario("Brasil", 21, 18, 10);
```

```
if (horario is object) Console.WriteLine("horario is object");
if (horario is Horario) Console.WriteLine("horario is Horario");
if (horario is FusoHorario) Console.WriteLine("horario is FusoHorario");
if (fusoHorario is object) Console.WriteLine("fusoHorario is object");
if (fusoHorario is Horario) Console.WriteLine("fusoHorario is Horario");
if (fusoHorario is FusoHorario) Console.WriteLine("fusoHorario is FusoHorario");
```

23. Chamada a método via classe base

```
Horario horario = new Horario(9, 10, 2);
Horario fusoHorario = new FusoHorario("Brasil", 9, 10, 2);
Console.WriteLine(horario.ToHora12());
Console.WriteLine(fusoHorario.ToHora12());
Console.ReadLine();
```

09:10:02 AM
09:10:02 AM

24. Polimorfismo

```
public class Horario : object
{
    ...

    public virtual string ToHora12()
    {
        ...
    }

    public virtual string ToHora24()
    {
        ...
    }
}

public class FusoHorario : Horario
{
    ...

    public override string ToHora12()
    {
        ...
    }

    public override string ToHora24()
    {
        ...
    }
}
```

09:10:02 AM
Brasil, 09:10:02 AM

25. Lidando com diferentes versões de classe

- 1º - Você utiliza uma biblioteca de terceiros (**Lib.dll**)... Esta biblioteca possui uma classe definida da seguinte forma:

```
class Imposto : object
{
    ...
}
```

- **Lib.dll** está na **versão 1.0.100.200** e encontra-se instalada no **GAC**.

26. Lidando com diferentes versões de classe

- 2º - Você resolve especializar a classe **Imposto**, adicionando uma nova funcionalidade:

```
class MeuImposto : Imposto
{
    ...

    public decimal ImpostoRenda()
    {
        ...
    }
}
```

27. Lidando com diferentes versões de classe

- 3º - Um ano mais tarde, a empresa que desenvolveu a biblioteca **Lib.dll**, resolve definir um novo método na classe **Imposto**... coincidentemente com a *mesma assinatura* que sua implementação na classe **MeuImposto**:

```
class Imposto : object
{
    ...

    public decimal ImpostoRenda()
    {
        ...
    }
}
```

- **Lib.dll** agora está na **versão 1.0.465.20** e encontra-se também instalada no **GAC**.

28. Lidando com diferentes versões de classe

- Eis agora o dilema... nossa aplicação cliente, obviamente utilizará a ***versão mais recente*** da Lib **1.0.***
- A chamada ao método **ImpostoRenda()** será feita a partir...
  - da classe **Imposto**?
  - ou da classe **MeuImposto**?
- Uma nova compilação de nosso projeto fará com que o compilador emita uma ***mensagem de advertência***... recomendando o uso da palavra chave **new**.
- O ideal neste caso, é, se possível, renomearmos o método para evitar possíveis problemas futuros.
- **MAS**... o mais importante a considerar é que a aplicação cliente ***continuará a funcionar normalmente*** mesmo após a instalação da nova biblioteca **Lib.dll**.

29. Liberando o objeto para a coleta de lixo prematuramente

```
Horario horario = new Horario(9, 10, 2);
Horario fusoHorario = new FusoHorario("Brasil", 9, 10, 2);
Console.WriteLine(horario.ToHora12());
Console.WriteLine(fusoHorario.ToHora12());
horario = null;
fusoHorario = null;
Console.ReadLine();
```

30. Métodos (passagem de parâmetros)

- Em C#... tudo é passado por valor...

```
static void Main(string[] args)
{
    Console.WriteLine(Soma(2, 3));
```

```
    string msg = "Hello";
    Show(msg);
}

static int Soma(int x, int y)
{
    return x + y;
}

static void Show(string message)
{
    Console.WriteLine(message);
}
```

31. Métodos (passagem de parâmetros)

- Em C#... tudo é passado por valor...
  - a não ser quando **ref** ou **out** é especificado.

```
static void Main(string[] args)
{
    int i = 5;
    GetNext(ref i);
    Console.WriteLine(i);

    int quociente, resto;
    Divide(10, 3, out quociente, out resto);
    Console.WriteLine(quociente);
    Console.WriteLine(resto);
}

static void GetNext(ref int numero)
{
    numero++;
}

static void Divide(int dividendo, int divisor, out int quociente, out int resto)
{
    quociente = dividendo / divisor;
    resto = dividendo % divisor;
}
```

32. Métodos (params)

- Em C#... é possível definir um número arbitrário de parâmetros...

```
static void Main(string[] args)
{
    MostraTotal("Livros", 2, 5, 7, 1);
}

static void MostraTotal(string mensagem, params int[] numeros)
{
    int tot = 0;
    foreach (int numero in numeros)
    {
        tot += numero;
    }
    Console.WriteLine("{0} = {1}", mensagem, tot);
}
```

33. Métodos (overload)

- Vários métodos com o mesmo nome... mas com assinaturas diferentes.

```
static int Soma(int x, int y)
{
    return x + y;
}
static int Soma(int x, int y, int z)
{
    return x + y + z;
}
static long Soma(int x, int y, int z) // inválido
{
    return x + y + z;
}
```

34. is (verificando tipo antes de fazer casting)

```
static void Main(string[] args)
{
    FusoHorario fuso = new FusoHorario("Brasil", 10, 20, 30);
    string str = "C#";
    int num = 1;
    ShowString(fuso);
    ShowString(str);
    ShowString(num);
    Console.ReadLine();
}

static void ShowString(object obj)
{
    if (obj is Horario)
    {
        Horario h = (Horario)obj;
        Console.WriteLine(h.ToHora12());
    }
    else
    {
        Console.WriteLine(obj.ToString());
    }
}
```

35. utilizando as...

```
static void ShowString(object obj)
{
    Horario h = obj as Horario;
    if (h != null)
    {
        Console.WriteLine(h.ToHora12());
    }
    else
    {
        Console.WriteLine(obj.ToString());
    }
}

Escrever... Horario h = obj as Horario;
é o mesmo que... Horario h = obj is Horario ? (Horario)obj : (Horario)null;
```

36. Overriding ToString()...

```
public class Horario : object
{
    ...
```

```
    public override string ToString()
    {
        return ToHora12();
    }
}

...

static void ShowString(object obj)
{
    Console.WriteLine(obj.ToString());
}
```

37. Membros de instância X Membros estáticos

- Membros de instância pertencem ao objeto instanciado.
  - Cada objeto instanciado possui seu próprio conjunto de dados.
  - Quando não especificamos a palavra chave ***static***, definimos um membro de instância.
- Membros estáticos pertencem a classe, não importando se há ou não a existência de uma instância da classe.
  - Cada classe possui um e apenas um conjunto de dados estáticos.
  - Quando especificamos a palavra chave ***static***, definimos um membro estático, ou seja, um membro de classe.
  - Um membro estático deve processar *somente dados estáticos*, a não ser é claro que uma referência a um objeto seja passada durante a chamada.

38. Tempo decorrido entre uma hora e outra...

```
Horario inicio = new Horario(16, 30, 0);
Horario termino = new Horario(20, 0, 0);
Horario duracao = termino - inicio;
Console.WriteLine(duracao.ToHora24());
Console.WriteLine(Horario.Subtract(termino, inicio).ToHora24());
```

39. Tempo decorrido entre uma hora e outra...

```
public class Horario : object
{
    ...

    public static Horario Subtract(Horario horaFinal, Horario horaInicial)
    {
        return horaFinal - horaInicial;
    }

    public static Horario operator -(Horario horaFinal, Horario horaInicial)
    {
        int segundosFinal = horaFinal.Hora * 3600 +
            horaFinal.Minuto * 60 + horaFinal.Segundo;
        int segundosInicial = horaInicial.Hora * 3600 +
            horaInicial.Minuto * 60 + horaInicial.Segundo;
        if (segundosFinal < segundosInicial)
        {
            return new Horario();
        }
        int segundos = segundosFinal - segundosInicial;
        byte hora = (byte)(segundos / 3600);
        byte minuto = (byte)((segundos - hora * 3600) / 60);
        byte segundo = (byte)(segundos - (hora * 3600 + minuto * 60));
        return new Horario(hora, minuto, segundo);
    }
}
```

40. Operadores

```
public class Numero
{
    int numero;

    public Numero(int numero)
    {
        this.numero = numero;
    }

    public static implicit operator Numero(int x)
    {
        return new Numero(x);
    }

    public static explicit operator int(Numero x)
    {
        return x.numero;
    }

    public static Numero operator +(Numero x, Numero y)
    {
        return new Numero(x.numero + y.numero);
    }
}
```

Utilizando...

```
Numero a = new Numero(1);
Numero b = 2;
int c = (int)b;
Numero d = a + b;
```

41. Classes Abstratas

- Uma classe abstrata é uma classe projetada com o propósito de ser *utilizada apenas como uma classe base*.
- Classes abstratas não podem ser instanciadas.

```
public abstract class ComponenteVisual
{
    int largura, altura;
    ...
}

public class Botao2D : ComponenteVisual
{
    ...
}

public class Botao3D : ComponenteVisual
{
    ...
}
```

42. Métodos abstratos

- Um método abstrato define um ***membro virtual***, porém, ***sem implementação***.
- Um método abstrato só pode ser definido numa classe que foi, também, declarada como **abstract**.
- Toda classe concreta derivada da classe que definiu o método abstrato, deve, obrigatoriamente, criar um corpo para este método.

```
public abstract class ComponenteVisual
```

```
{
    int largura, altura;

    public abstract void Desenha();
}

public class Botao2D : ComponenteVisual
{
    public override void Desenha()
    {
        ...
    }
}

public class Botao3D : ComponenteVisual
{
    // erro: a classe deve, obrigatoriamente, implementar Desenha().
}
```

43. Métodos externos

- Um método externo é um cabeçalho de método que faz referência a uma definição de método disponível em uma biblioteca (**dll**) escrita em uma linguagem não .net.
- Pode ser utilizado para fazer chamadas a **API** do Windows.

```
using System.Security.Permissions;
using System.Runtime.InteropServices;

[DllImport("Kernel32.dll")]
public static extern bool CreateDirectory(string name, SecurityAttribute sa);

CreateDirectory(@"C:\Temp\X3", null);
```

44. Classes e Membros Selados

- Uma classe selada (**sealed**) é uma classe que *não permite* que outras classes derivem dela.
- Um membro selado (por exemplo, um método) é um membro que ***não permite a sua sobrescrita*** (**override**) numa classe derivada da classe onde ele foi declarado como **sealed**.
- O compilador C# faz ***otimizações*** em chamadas a métodos virtuais da classe selada, em virtude desta, com certeza, não possuir derivações.
- **C# x Java...**
  - **C#...**
    - Existe membro **virtual**.
    - Existe membro **não virtual**.
    - Membros não virtuais podem ser colocados **inline** pelo compilador...
      - ⇒ Melhora o *desempenho*.
      - ⇒ Chamada *otimizada*.
      - ⇒ Otimização feita durante a *compilação (fase JIT)*.
      - ⇒ Métodos e classes **sealed** também usufruem desse processo.
  - **Java...**
    - *Todo membro* definido na classe é considerado como **virtual**.
    - *Nenhuma otimização pode ser feita para estes membros.*

45. Classes e Membros Selados

```
public abstract class ComponenteVisual
{
    int largura, altura;

    public abstract void Desenha();
}

public sealed class Botao2D : ComponenteVisual
```

```
{
    public override void Desenha()
    {
        ...
    }
}

public class BotaoTriangular : Botao2D
{
    // erro: classe Botao2D está selada.
}
```

46. Nested Types (Tipos Aninhados)

- Um “nested type” é um tipo declarado dentro de outro tipo.

```
public sealed class Botao2D : ComponenteVisual
{
    public override void Desenha()
    {
        Dimensao d = new Dimensao();
        // a classe Dimensao tem propósitos apenas
        // dentro de Botao2D... e em nenhum outro
        // lugar mais.
    }

    class Dimensao // Nested type
    {
        // implementação de Dimensao
    }
}
```

47. Constantes

- Uma constante é um membro da classe que possui um ***valor imutável***.
- As constantes devem ser inicializadas em sua declaração.
- As constantes são ***implicitamente estáticas***, porém, não devem ser declaradas com o modificador **static**.

```
public class Funcionario
{
    public const decimal SalarioMinimo = 1000M;

    ...
}
```

48. Campos readonly

- Um campo **readonly** comporta-se, de certa forma, como uma constante...
- Um campo **readonly** pode ser inicializado tanto na sua ***declaração*** quanto em um dos ***construtores*** da classe.
- Campos **readonly** podem ser ***estáticos*** ou de ***instâncias***.

```
public class Usuario
{
    readonly int nivelAcesso;

    public Usuario()
    {
        nivelAcesso = 1; // ou obtem o nível de acesso
                         // a partir de uma consulta no
                         // banco de dados.
    }
}
```

49. Indexadores

- Permitem tratarmos as classes ***sintaticamente*** como se fossem *arrays*.
  - Ideal para classes que contenham *conjunto de dados* (coleções).

```
public class Notas
{
    double[] notas;

    public Notas(params double[] notas)
    {
        this.notas = new double[notas.Length];
        for (int i = 0; i < notas.Length; i++)
        {
            this.notas[i] = notas[i];
        }
    }

    public double Media()
    {
        double total = 0D;
        foreach (double nota in notas)
        {
            total += nota;
        }
        return total / notas.Length;
    }

    public double this[int index]
    {
        get
        {
            return notas[index];
        }
        set
        {
            notas[index] = value;
        }
    }
}
```

Utilizando...

```
Notas notas = new Notas(9D, 5.8, 10D);
Console.WriteLine("Nota 1 = {0}", notas[0]);
Console.WriteLine("Nota 2 = {0}", notas[1]);
Console.WriteLine("Nota 3 = {0}", notas[2]);
Console.WriteLine(notas.Media());
```

50. typeof

```
Horario h = new FusoHorario("Brasil", 21, 13, 10);

if (h.GetType() == typeof(Horario))
{
    Console.WriteLine("h é uma instância de Horario."); // Não exibe
}

if (h.GetType() == typeof(FusoHorario))
{
    Console.WriteLine("h é uma instância de FusoHorario."); // Exibe
}
```

51. Gerenciamento de Memória

```
Numero obj1 = new Numero(5);
Numero obj2 = new Numero(5);
if (obj1 != obj2) Console.WriteLine("obj1 != obj2"); // Exibe
Numero obj3 = obj1;
if (obj3 == obj1) Console.WriteLine("obj3 == obj1"); // Exibe

string x, y;
x = "C# Language";
y = "C# Language"; // Há uma otimização do compilador neste ponto
                   // para a string anterior.
if (x == y) Console.WriteLine("x == y"); // Exibe
// O operador == está sobrecarregado para comparar o conteúdo
// e não o endereço dos objetos.
```

52. Construtores de instância e estáticos

- Construtores de instância podem ser sobrecarregados.
- Só pode haver ***um construtor estático*** por classe...

```
static NomeClasse()
{
    // inicialização de dados estáticos somente
}
```

53. Construtores

```
public class Classe1 : object
{
    int x, y;

    public Classe1(int x, int y) : base()
    {
        this.x = x;
        this.y = y;
    }

    public Classe1(int x) : this(x, -1)
    {
    }

    public Classe1() : this(-1, -1)
    {
    }
}
```

54. Construtores

```
public class Classe2 : Classe1
{
    int z;

    public Classe2(int x, int y, int z) : base(x, y)
    {
        this.z = z;
    }

    public Classe2(int x, int y) : this(x, y, -1)
    {
    }

    public Classe2(int x) : this(x, -1, -1)
    {
```

```
    }

    public Classe2() : this(-1, -1, -1)
    {
    }
}
```

55. Construtores

```
public class Classe1 : object
{
    int x, y;

    public Classe1(int x, int y) : base()
    {
        Console.WriteLine("Classe1(int x, int y)");
        this.x = x;
        this.y = y;
    }

    public Classe1(int x) : this(x, -1)
    {
        Console.WriteLine("Classe1(int x)");
    }

    public Classe1() : this(-1, -1)
    {
        Console.WriteLine("Classe1()");
    }
}
```

56. Construtores

```
public class Classe2 : Classe1
{
    int z;

    public Classe2(int x, int y, int z) : base(x, y)
    {
        Console.WriteLine("Classe2(int x, int y, int z)");
        this.z = z;
    }

    public Classe2(int x, int y) : this(x, y, -1)
    {
        Console.WriteLine("Classe2(int x, int y)");
    }

    public Classe2(int x) : this(x, -1, -1)
    {
        Console.WriteLine("Classe2(int x)");
    }

    public Classe2() : this(-1, -1, -1)
    {
        Console.WriteLine("Classe2()");
    }
}
```

57. Construtores (sequência de chamada)

```
Console.WriteLine("a...");
Classe1 a = new Classe1(); // Classe1(int x, int y)
                           // Classe1()
```

```
Console.WriteLine("b...");
Classe1 b = new Classe1(5); // Classe1(int x, int y)
                            // Classe1(int x)

Console.WriteLine("c...");
Classe1 c = new Classe1(5, 6); // Classe1(int x, int y)
```

58. Construtores (sequência de chamada)

```
Console.WriteLine("d...");
Classe2 d = new Classe2(); // Classe1(int x, int y)
                           // Classe2(int x, int y, int z)
                           // Classe2()

Console.WriteLine("e...");
Classe2 e = new Classe2(1); // Classe1(int x, int y)
                            // Classe2(int x, int y, int z)
                            // Classe2(int x)

Console.WriteLine("f...");
Classe2 f = new Classe2(1, 2); // Classe1(int x, int y)
                               // Classe2(int x, int y, int z)
                               // Classe2(int x, int y)

Console.WriteLine("g...");
Classe2 g = new Classe2(1, 2, 3); // Classe1(int x, int y)
                                  // Classe2(int x, int y, int z)
```

59. Destrutores

- Os objetos no ***heap*** são removidos pelo *coletor de lixo* (**garbage collector**), e pode haver uma demora entre o momento em que o objeto torna-se inacessível e a destruição real pelo *coletor de lixo*.
- Os destrutores, na arquitetura subjacente do .net, correspondem ao método **Finalize()**.

```
public class Classe1 : object
{
    ...

    ~Classe1()
    {
        Console.WriteLine("~Classe1()");
    }
}

public class Classe2 : Classe1
{
    ...

    ~Classe2()
    {
        Console.WriteLine("~Classe2()");
    }
}
```

60. Destrutores (sequência de chamada)

- *Exemplo 1...*

```
Classe1 a = new Classe1();
// Classe1(int x, int y)
// Classe1()
// ~Classe1();
```

- *Exemplo 2...*

```
Classe2 a = new Classe2();
// Classe1(int x, int y)
// Classe2(int x, int y, int z)
// Classe2()
// ~Classe2()
// ~Classe1()
```

61. Destrutores + IDisposable

- Mas e se os objetos utilizarem ***recursos escassos*** do sistema...?
  - File Handles.
  - Controladores GDI.
  - Conexões com o banco de dados.
  - ***Solução***...
    - A classe deve implementar a interface **IDisposable**.
    - Um ***destrutor*** deve ser definido.

Estudaremos esta funcionalidade numa outra aula...

62. OOP...

- Dúvidas?

1. Estruturas

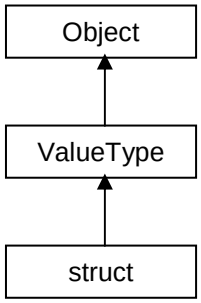
2. Keywords...

[✓] abstract	[ ] event	[✓] new	[✓] struct
[✓] as	[✓] explicit	[✓] null	[✓] switch
[✓] base	[✓] extern	[✓] object	[✓] this
[✓] bool	[✓] false	[✓] operator	[ ] throw
[✓] break	[ ] finally	[✓] out	[✓] true
[✓] byte	[ ] fixed	[✓] override	[ ] try
[✓] case	[✓] float	[✓] params	[✓] typeof
[ ] catch	[✓] for	[✓] private	[✓] uint
[✓] char	[✓] foreach	[✓] protected	[✓] ulong
[✓] checked	[✓] goto	[✓] public	[✓] unchecked
[✓] class	[✓] if	[✓] readonly	[ ] unsafe
[✓] const	[✓] implicit	[✓] ref	[✓] ushort
[✓] continue	[✓] in	[✓] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[✓] virtual
[✓] default	[ ] interface	[✓] sealed	[ ] volatile
[ ] delegate	[✓] internal	[✓] short	[✓] void
[✓] do	[✓] is	[ ] sizeof	[✓] while
[✓] double	[ ] lock	[ ] stackalloc	
[✓] else	[✓] long	[✓] static	
[ ] enum	[✓] namespace	[✓] string	

3. Estruturas

- Classes são *tipo referência* (heap).
- Estruturas são *tipo valor* (pilha ou ***inline***)...
  - Ideal para ocasiões em que precisamos de algo muito parecido com as classes, mas sem o custo envolvido na criação de objetos no heap.
  - ***Não suportam herança***.
  - O compilador sempre fornece um **construtor vazio** (sem nenhum parâmetro), que nunca poderá ser substituído.
    - Devido a *problemas de implementação* no runtime do .net, a Microsoft optou por proibir a criação de construtores sem parâmetros para estruturas... eles sempre existirão e você não poderá fazer nada para modificá-los ou mesmo, ocultá-los.

```
struct Point
{
    int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```



4. Modificadores de Acesso

```
namespace ...
{
```

```
internal
public   struct ...
{
    ...
    private
    public
    internal              Constructor(...)
    protected
    protected internal
    {
        ...
    }
}
}
```

5. Estruturas x Classes

- Membros de uma estrutura...
  - **Dados (membros de armazenamento)**...
    - Constantes.
    - Campos.
    - Eventos.
  - **Funções (membros que contém código)**...
    - Construtores.
    - ~~Destrutor~~.
    - Propriedades.
    - Métodos.
    - Indexadores.
    - Operadores.
  - Outros tipos (**nested types**)...
    - Classes, estruturas, etc.

6. Point

```
struct Point
{
    int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public int X
    {
        get
        {
            return x;
        }
        set
        {
            x = value;
        }
    }

    public int Y
    {
        get
```

```
        {
            return y;
        }
        set
        {
            y = value;
        }
    }

    public override string ToString()
    {
        return string.Format("({0}, {1})", x, y);
    }
}
```

Utilizando...

```
Point p1 = new Point(1, 2);
Point p2 = p1;
p1.X = 10;
Console.WriteLine(p1.ToString()); // (10, 2)
Console.WriteLine(p2.ToString()); // (1, 2)
```

7. Limitações das Estruturas

- Não é possível ***inicializar campos*** durante sua declaração (a não ser ***campos estáticos*** e ***constantes***).
- Não suportam ***herança de implementação***... mas permitem ***herança de interface***.
- Estruturas, por serem tipo valor, não podem receber **null**.
- O operador de igualdade (==) não pode ser utilizado para comparar estruturas.
  - if (p1 == p2) ... // erro
  - if (p1.Equals(p2)) ... // certo

```
struct Point
{
    int x = 1, y = 2; // erro
    ...
}

Point p1 = new Point(1, 2);
Point p2 = new Point(1, 2);
if (p1.Equals(p2)) Console.WriteLine("p1 == p2");
```

8. Classes x Estruturas (A diferença entre a vida e a morte)

```
struct Point
{
    int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Utilizando...

```
Point[] points = new Point[100];
for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
// 1 objeto do tipo "Array de Points" com 100 elementos é criado.
// Cada elemento no array contém o par de dados x, y.
// Desta forma temos apenas 1 objeto alocado no Heap.
```

```
// Este objeto já contem os 100 elementos.
// Os elementos do objeto já contem os dados (inline).
```

9. Classes x Estruturas (A diferença entre a vida e a morte)

```
class Point
{
    int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Utilizando...

```
Point[] points = new Point[100];
for (int i = 0; i < 100; i++) points[i] = new Point(i, i);
// 1 objeto do tipo "Array de Points" com 100 ponteiros é criado e
// 100 objetos da classe Point são criados e alocados no Heap.
// Desta forma temos 101 objetos alocado no Heap.
// 100 objetos contendo os valores x, y.
// 1 único objeto contendo 100 ponteiros para os elementos
// de dados x, y.
```

10. Boxing / Unboxing

```
Point p = new Point(10, 10);
object obj = p;
p.x = 20;
Console.WriteLine(((Point)obj).x);

struct Point
{
    public int x, y;

    public Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}
```

Se Point for uma estrutura então o resultado será 10, mas se for uma classe, então o resultado será 20.

11. C#

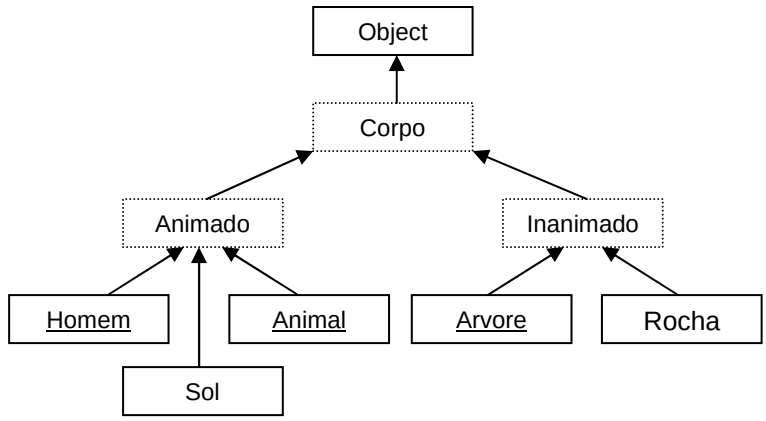
- Dúvidas

1. Interfaces

2. Keywords...

[✓] abstract	[ ] event	[✓] new	[✓] struct
[✓] as	[✓] explicit	[✓] null	[✓] switch
[✓] base	[✓] extern	[✓] object	[✓] this
[✓] bool	[✓] false	[✓] operator	[ ] throw
[✓] break	[ ] finally	[✓] out	[✓] true
[✓] byte	[ ] fixed	[✓] override	[ ] try
[✓] case	[✓] float	[✓] params	[✓] typeof
[ ] catch	[✓] for	[✓] private	[✓] uint
[✓] char	[✓] foreach	[✓] protected	[✓] ulong
[✓] checked	[✓] goto	[✓] public	[✓] unchecked
[✓] class	[✓] if	[✓] readonly	[ ] unsafe
[✓] const	[✓] implicit	[✓] ref	[✓] ushort
[✓] continue	[✓] in	[✓] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[✓] virtual
[✓] default	[✓] interface	[✓] sealed	[ ] volatile
[ ] delegate	[✓] internal	[✓] short	[✓] void
[✓] do	[✓] is	[ ] sizeof	[✓] while
[✓] double	[ ] lock	[ ] stackalloc	
[✓] else	[✓] long	[✓] static	
[ ] enum	[✓] namespace	[✓] string	

3. O jogo



Classes Abstratas

Objetos que morrem interface: **IMortal**

4. Implementação

```
abstract class Corpo : object
{
    int x, y, z; // posição no espaço
    ...
```

```
}

abstract class Animado : Corpo
{
    void MoveTo(int x, int y, int z)
    {
        ...
    }
}

abstract class Inanimado : Corpo
{
}
```

5. Implementação

```
interface IMortal
{
    void Die();
}

class Homem : Animado, IMortal
{
    int idade;

    public void Die()
    {
        if (idade > 99)
        {
            ...
        }
    }
}

class Arvore : Inanimado, IMortal
{
    int nivelNutrienteSolo;

    public void Die()
    {
        if (nivelNutrienteSolo < 9)
        {
            ...
        }
    }
}
```

6. Aplicabilidade

```
static void KillAll(object[] objs)
{
    foreach (object obj in objs)
    {
        if (obj is IMortal)
        {
            ((IMortal)obj).Die();
        }
    }
}
```

ou... (sem interfaces)

```
static void KillAll(object[] objs)
{
```

```
    foreach (object obj in objs)
    {
        if (obj is Homem)
        {
            ((Homem)obj).Die();
        }
        if (obj is Arvore)
        {
            ((Arvore)obj).Die();
        }
        ...
    }
}
```

7. Interface

- Por padrão, utiliza-se ***notação húngara*** (I...) para nomeação de interfaces.
- Uma interface define um ***contrato***.
- Uma classe ou estrutura, que "**herda**" uma interface, de fato, **assina um contrato com o compilador** dizendo que irá, *implicita* ou *explicitamente*, implementar todos os membros declarados na interface.
- Uma classe ou estrutura pode implementar *uma ou mais* interfaces.
- Uma interface também pode herdar de outras interfaces.
- Uma interface pode conter...
  - *Propriedades*.
  - *Métodos*.
  - *Eventos*.
  - *Indexadores*.
- Tudo o que uma interface contem são ***assinaturas*** para seus membros.
- Os membros da interface são **sempre públicos**.

8. Exemplo

```
interface IControl
{
    void Paint();
}

interface ITextBox : IControl
{
    void SetText(string text);
}

interface IListBox : IControl
{
    void SetItems(string[] items);
}

interface IComboBox : ITextBox, IListBox
{
}

class MyComboBox : IComboBox
{
    public void Paint()
    {
    }

    public void SetText(string text)
    {
    }

    public void SetItems(string[] items)
```

```
    {
    }
}
```

9. Exemplo

```
interface IStringList
{
    void Add(string text);
    bool Remove(string text);

    int Count
    {
        get;
    }

    string this[int index]
    {
        get;
        set;
    }
}
```

10. Implementação Explícita

```
interface IInterface1
{
    void DoSomething();
}

interface IInterface2
{
    void DoSomething();
}

class MyClass : IInterface1, IInterface2
{
    void IInterface1.DoSomething()
    {
        Console.WriteLine("Do by Interface 1");
    }

    void IInterface2.DoSomething()
    {
        Console.WriteLine("Do by Interface 2");
    }
}
```

invocando...

```
MyClass mc = new MyClass();
((IInterface1)mc).DoSomething();
((IInterface2)mc).DoSomething();
```

11. Implementação Explícita

```
class MyClass : IInterface1, IInterface2
{
    void IInterface1.DoSomething()
    {
        Console.WriteLine("Do by Interface 1");
    }

    void IInterface2.DoSomething()
```

```
    {
        Console.WriteLine("Do by Interface 2");
    }

    public void DoSomething(byte n)
    {
        if (n == 1)
        {
            ((IInterface1)this).DoSomething();
        }
        else
        {
            ((IInterface2)this).DoSomething();
        }
    }
}
```

invocando...

```
MyClass mc = new MyClass();
mc.DoSomething(1);
```

12. C#

- Dúvidas?

1. Enumerações

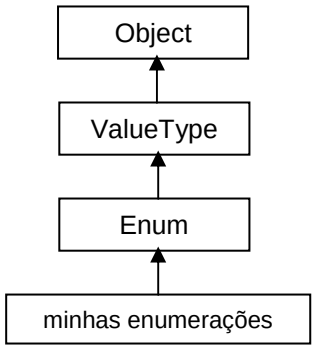
2. Keywords...

[✓] abstract	[ ] event	[✓] new	[✓] struct
[✓] as	[✓] explicit	[✓] null	[✓] switch
[✓] base	[✓] extern	[✓] object	[✓] this
[✓] bool	[✓] false	[✓] operator	[ ] throw
[✓] break	[ ] finally	[✓] out	[✓] true
[✓] byte	[ ] fixed	[✓] override	[ ] try
[✓] case	[✓] float	[✓] params	[✓] typeof
[ ] catch	[✓] for	[✓] private	[✓] uint
[✓] char	[✓] foreach	[✓] protected	[✓] ulong
[✓] checked	[✓] goto	[✓] public	[✓] unchecked
[✓] class	[✓] if	[✓] readonly	[ ] unsafe
[✓] const	[✓] implicit	[✓] ref	[✓] ushort
[✓] continue	[✓] in	[✓] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[✓] virtual
[✓] default	[✓] interface	[✓] sealed	[ ] volatile
[ ] delegate	[✓] internal	[✓] short	[✓] void
[✓] do	[✓] is	[ ] sizeof	[✓] while
[✓] double	[ ] lock	[ ] stackalloc	
[✓] else	[✓] long	[✓] static	
[✓] enum	[✓] namespace	[✓] string	

3. Enumeração

- Uma enumeração é um **tipo valor** que declara um ***conjunto de constantes (membros)***.
- As enumerações são, de fato, estruturas derivadas da classe base **System.Enum**.
- Toda enumeração possui um tipo subjacente (*underlying type*).
- Quando o tipo subjacente não é explicitamente especificado, o compilador C# considera como sendo **int**.
- Os tipos podem ser...
  - byte, sbyte
  - short, ushort
  - int, uint
  - long, ulong

```
enum Color : int
{
    Red,
    Green,
    Blue
}
```



4. Enumeração

```
enum Faixa // Faixa = int
{
    Branca,    // Branca   =  0
    Amarela,   // Amarela  =  1
    Vermelha,  // Vermelha =  2
    Laranja,   // Laranja  =  3
```

```
    Verde,     // Verde    =  4
    Roxa = 10, // Roxa     = 10
    Marrom,    // Marrom   = 11
    Preta = 3  // Preta    =  3
}
```

utilizando...

```
Faixa faixa = Faixa.Marrom;
Console.WriteLine((int)faixa); // 11
Console.WriteLine(faixa.ToString()); // Marrom
```

5. Flags

```
public enum EnumSemFlags
{
    A = 1,
    B = 2,
    //  3 = 3
    C = 4,
    //  5 = 5
    //  6 = 6
    //  7 = 7
    D = 8
    //  9 = 9
    // 10 = 10
    // 11 = 11...
}

[Flags]
public enum EnumComFlags
{
    A = 1,
    B = 2,
    //  3 = A, B
    C = 4,
    //  5 = A, C
    //  6 = B, C
    //  7 = A, B, C
    D = 8
    //  9 = A, D
    // 10 = B, D
    // 11 = A, B, D
}
```

utilizando...

```
EnumSemFlags esf = (EnumSemFlags)3;
Console.WriteLine(esf.ToString()); // 3
EnumComFlags ecf = (EnumComFlags)3;
Console.WriteLine(ecf.ToString()); // A, B
```

6. Flag booleano

```
[Flags]
enum Permissao
{
    Read = 1,
    Write = 2,
    ReadWrite = Read | Write,
    Copy = 4,
    Delete = 8
}
```

utilizando...

```
Permissao usuario = Permissao.Copy | Permissao.ReadWrite;

if ((usuario & Permissao.Copy) == Permissao.Copy)
{
    Console.WriteLine("Permissão de cópia.");
}
if ((usuario & Permissao.Read) == Permissao.Read)
{
    Console.WriteLine("Permissão de leitura.");
}
if ((usuario & Permissao.Write) == Permissao.Write)
{
    Console.WriteLine("Permissão de gravação.");
}
```

7. C#

- Dúvidas?

1. Delegações e Eventos

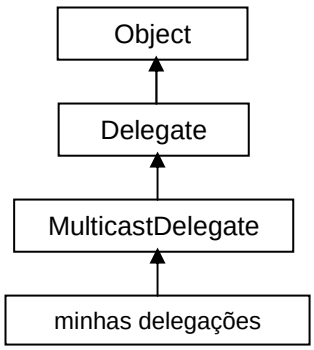
2. Keywords...

[✓] abstract	[✓] event	[✓] new	[✓] struct
[✓] as	[✓] explicit	[✓] null	[✓] switch
[✓] base	[✓] extern	[✓] object	[✓] this
[✓] bool	[✓] false	[✓] operator	[ ] throw
[✓] break	[ ] finally	[✓] out	[✓] true
[✓] byte	[ ] fixed	[✓] override	[ ] try
[✓] case	[✓] float	[✓] params	[✓] typeof
[ ] catch	[✓] for	[✓] private	[✓] uint
[✓] char	[✓] foreach	[✓] protected	[✓] ulong
[✓] checked	[✓] goto	[✓] public	[✓] unchecked
[✓] class	[✓] if	[✓] readonly	[ ] unsafe
[✓] const	[✓] implicit	[✓] ref	[✓] ushort
[✓] continue	[✓] in	[✓] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[✓] virtual
[✓] default	[✓] interface	[✓] sealed	[ ] volatile
[✓] delegate	[✓] internal	[✓] short	[✓] void
[✓] do	[✓] is	[ ] sizeof	[✓] while
[✓] double	[ ] lock	[ ] stackalloc	
[✓] else	[✓] long	[✓] static	
[✓] enum	[✓] namespace	[✓] string	

3. delegate

- As delegações são "*parecidas*" com ponteiros para funções.
- No entanto, as delegações também são **totalmente orientadas a objetos**.
- Uma delegação encapsula *dados do método a ser invocado*, bem como também se este é um método de *instância* ou *estático*.
- Uma delegação pode referenciar "apontar" *mais de um método* no mesmo objeto ou em objetos diferentes, desde que todos possuam a **mesma assinatura e tipo de retorno**.
- Quando uma *instância da delegação* é invocada, todas as entidades referenciadas por ela serão invocadas com os mesmos argumentos passados para a delegação.

```
delegate void SomeMethod();
delegate int Calcula(int x, int y);
```



4. delegate

```
class Program
{
    delegate void SomeMethod();
```

```
    static void Main(string[] args)
    {
        SomeMethod some = new SomeMethod(Do1);
        some();
    }

    static void Do1()
    {
        Console.WriteLine("Do1");
    }
}
```

5. delegate

```
class Program
{
    delegate void SomeMethod();

    static void Main(string[] args)
    {
        SomeMethod some = new SomeMethod(Do1);
        some += new SomeMethod(Do2);
        some();
    }

    static void Do1()
    {
        Console.WriteLine("Do1");
    }

    static void Do2()
    {
        Console.WriteLine("Do2");
    }
}
```

6. delegate

```
class Program
{
    delegate void Calcula(int x, int y);

    static void Main(string[] args)
    {
        Calcula calcula = new Calcula(Soma);
        calcula += new Calcula(Multiplica);
        calcula(2, 3);
    }

    static void Soma(int x, int y)
    {
        Console.WriteLine(x + y);
    }

    static void Multiplica(int x, int y)
    {
        Console.WriteLine(x * y);
    }
}
```

7. delegate

```
class Program
```

```
{
    delegate void Calcula(int x, int y);

    static void Main(string[] args)
    {
        Calcula c1 = new Calcula(Soma);
        Calcula c2 = new Calcula(Multiplica);
        Calcula calcula = c1 + c2;
        calcula(2, 3);
    }

    static void Soma(int x, int y)
    {
        Console.WriteLine(x + y);
    }

    static void Multiplica(int x, int y)
    {
        Console.WriteLine(x * y);
    }
}
```

8. delegate

```
delegate void Metodo(int numero);

class Classe1
{
    public static void M1(int i)
    {
        Console.WriteLine("Classe1.M1: " + i);
    }

    public void M2(int i)
    {
        Console.WriteLine("Instance.M2: " + i);
    }
}
```

utilizando...

```
Metodo d1 = new Metodo(Classe1.M1);
d1(1); // Classe1.M1: 1
Classe1 c = new Classe1();
Metodo d2 = new Metodo(c.M2);
d2(2); // Instance.M2: 2
Metodo d3 = d1 + d2;
d3(3); // Classe1.M1: 3, Instance.M2: 3
d3 += d1;
d3(4); // Classe1.M1: 4, Instance.M2: 4, Classe1.M1: 4
d3 -= d1; // Remove última referência a Classe1.M1
```

9. Eventos

```
public class SaldoNaoDisponivelEventArgs : EventArgs
{
    decimal saldoAtual;
    decimal valorRequisitado;

    public SaldoNaoDisponivelEventArgs(
           decimal saldoAtual,
           decimal valorRequisitado) : base()
    {
        this.saldoAtual = saldoAtual;
```

```
        this.valorRequisitado = valorRequisitado;
    }

    public decimal SaldoAtual
    {
        get
        {
            return saldoAtual;
        }
    }

    public decimal ValorRequisitado
    {
        get
        {
            return valorRequisitado;
        }
    }
}
```

10. Eventos

```
public delegate void SaldoNaoDisponivelEventHandler(object sender,
SaldoNaoDisponivelEventArgs e);

public class Conta : object
{
    decimal saldo;

    public Conta(decimal saldo) : base()
    {
        this.saldo = saldo;
    }

    public event SaldoNaoDisponivelEventHandler SaldoNaoDisponivel;

    public void Debita(decimal valor)
    {
        if (valor <= saldo)
        {
            saldo -= valor;
        }
        else
        {
            OnSaldoNaoDisponivel(saldo, valor);
        }
    }

    protected virtual void OnSaldoNaoDisponivel(decimal saldo, decimal valor)
    {
        if (SaldoNaoDisponivel != null)
        {
            SaldoNaoDisponivel(this,
                new SaldoNaoDisponivelEventArgs(saldo, valor));
        }
    }
}
```

11. Eventos

```
class Program
{
    static void Main(string[] args)
    {
```

```
        Conta conta = new Conta(1000M);
        conta.SaldoNaoDisponivel += new
              SaldoNaoDisponivelEventHandler(conta_SaldoNaoDisponivel);
        conta.Debita(200); // Restam 800
        conta.Debita(500); // Restam 300
        conta.Debita(400); // Gera evento
    }

    static void conta_SaldoNaoDisponivel(object sender,
                SaldoNaoDisponivelEventArgs e)
    {
        Console.WriteLine("Saldo atual = {0:C}\nValor requisitado = {1:C}",
            e.SaldoAtual,
            e.ValorRequisitado);
    }
}
```

12. C#

- Dúvidas?

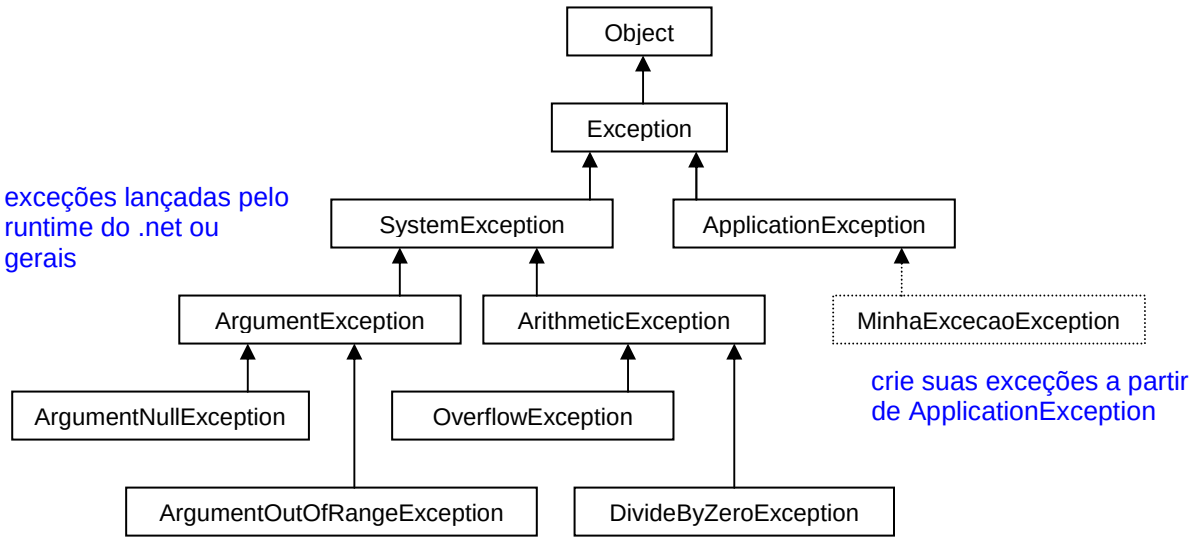
1. Exceções

2. Keywords...

[✓] abstract	[✓] event	[✓] new	[✓] struct
[✓] as	[✓] explicit	[✓] null	[✓] switch
[✓] base	[✓] extern	[✓] object	[✓] this
[✓] bool	[✓] false	[✓] operator	[✓] throw
[✓] break	[✓] finally	[✓] out	[✓] true
[✓] byte	[ ] fixed	[✓] override	[✓] try
[✓] case	[✓] float	[✓] params	[✓] typeof
[✓] catch	[✓] for	[✓] private	[✓] uint
[✓] char	[✓] foreach	[✓] protected	[✓] ulong
[✓] checked	[✓] goto	[✓] public	[✓] unchecked
[✓] class	[✓] if	[✓] readonly	[ ] unsafe
[✓] const	[✓] implicit	[✓] ref	[✓] ushort
[✓] continue	[✓] in	[✓] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[✓] virtual
[✓] default	[✓] interface	[✓] sealed	[ ] volatile
[✓] delegate	[✓] internal	[✓] short	[✓] void
[✓] do	[✓] is	[ ] sizeof	[✓] while
[✓] double	[ ] lock	[ ] stackalloc	
[✓] else	[✓] long	[✓] static	
[✓] enum	[✓] namespace	[✓] string	

3. Exceção (Exception)

- Uma exceção é um objeto criado (***lançado***) quando uma determinada *condição de erro* em particular ocorre.
- Esse objeto contem **informações** que devem ajudar a ***rastrear o problema***.



4. try... catch... finally

```
try
{
    Bloco de código que pode, eventualmente, lançar uma
    exceção.
}
catch (Exception1 e)
{
    Tratamento mais específico para a exceção lançada.
}
catch (Exception2 e)
{
    Tratamento mais geral para a exceção lançada.
}
catch
{
    Tratamento de exceções lançadas por código
    não gerenciado.
}
finally
{
    Código de liberação.
}
```

5. try... catch

```
try
{
    byte b = 255;
    checked
    {
        b++;
    }
}
catch (OverflowException e)
{
    Console.WriteLine(
        "Message: {0}\nSource: {1}\nStackTrace: {2}\nTargetSite: {3}",
        e.Message, e.Source, e.StackTrace, e.TargetSite);
}
```

6. try... catch

```
try
{
    string textoDividendo, textoDivisor;
    byte dividendo, divisor;
    Console.Write("Dividendo: ");
    textoDividendo = Console.ReadLine();
    Console.Write("Divisor: ");
    textoDivisor = Console.ReadLine();
    dividendo = byte.Parse(textoDividendo);
    divisor = byte.Parse(textoDivisor);
    byte resultado = (byte)(dividendo / divisor);
    Console.WriteLine(resultado);
}
catch (OverflowException e)
{
    Console.WriteLine("OverflowException: " + e.Message);
}
catch (DivideByZeroException e)
{
    Console.WriteLine("DivideByZeroException: " + e.Message);
```

```
}
catch (FormatException e)
{
    Console.WriteLine("FormatException: " + e.Message);
}
```

7. try... catch

```
try
{
    string textoDividendo, textoDivisor;
    byte dividendo, divisor;
    Console.Write("Dividendo: ");
    textoDividendo = Console.ReadLine();
    Console.Write("Divisor: ");
    textoDivisor = Console.ReadLine();
    dividendo = byte.Parse(textoDividendo);
    divisor = byte.Parse(textoDivisor);
    byte resultado = (byte)(dividendo / divisor);
    Console.WriteLine(resultado);
}
catch (ArithmeticException e) // cobre Overflow e DivideByZero
{
    Console.WriteLine("ArithmeticException: " + e.Message);
}
catch (Exception e) // cobre todas as exceções gerenciadas
{
    Console.WriteLine("Exception: " + e.Message);
}
```

8. Lançando uma exceção

```
static void Main(string[] args)
{
    try
    {
        Console.WriteLine(Soma(1, 2, 3, 4, 5));
        Console.WriteLine(Soma(null));
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}

static int Soma(params int[] numeros)
{
    if (numeros == null)
    {
        throw new ArgumentNullException(
            "numeros",
            "Método Soma(), parâmetro não pode ser null."
            );
    }
    int tot = 0;
    foreach (int numero in numeros)
    {
        tot += numero;
    }
    return tot;
}
```

9. Exception

- Propriedades...
  - *Alimentadas automaticamente pelo runtime do .net...*
    - **Source**: Nome da aplicação (*montagem*) responsável pela exceção.
    - **StackTrace**: Rastreamento de chamada de método na pilha.
    - **TargetSite**: Objeto de reflexão que descreve o método que lançou a exceção.
  - *Você deve alimentar essas propriedades quando lançar as exceções...*
    - **Message**: Texto informativo.
    - **HelpLink**: Arquivo de ajuda.
    - **InnerException**: Quando uma exceção é lançada dentro de um bloco catch, esta propriedade conterá uma referência ao objeto exceção que desviou o fluxo de execução para o bloco catch.

10. Criando exceções

```
class MinhaAppException : ApplicationException
{
    // campos que armazenam informações sobre
    // as exceções de nossa aplicação.

    public MinhaAppException() : base("Exceção em MinhaApp.")
    {
    }

    public MinhaAppException(string message) : base(message)
    {
    }

    public MinhaAppException(string message, Exception innerException) :
           base(message, innerException)
    {
    }

    // propriedades de leitura/escrita dos campos
}
```

11. try... catch (global)

- O que o **runtime do .net** faz quando uma exceção não é tratada?

```
try
{
```

assembly

```
}
catch
{
```

mensagem

aborta execução

```
}
```

12. Blocos try aninhados

```
try
{
    // ponto A – o que ocorre se uma exceção é lançada neste ponto?
    try
    {
        // ponto B – e neste ponto?
    }
```

```
    catch
    {
        // ponto C
    }
    finally
    {
        // ponto D
    }
}
catch
{
    // ponto E
}
finally
{
    // ponto F
}
```

13. C#

- Dúvidas?

1. Código não seguro

2. Keywords...

[✓] abstract	[✓] event	[✓] new	[✓] struct
[✓] as	[✓] explicit	[✓] null	[✓] switch
[✓] base	[✓] extern	[✓] object	[✓] this
[✓] bool	[✓] false	[✓] operator	[✓] throw
[✓] break	[✓] finally	[✓] out	[✓] true
[✓] byte	[✓] fixed	[✓] override	[✓] try
[✓] case	[✓] float	[✓] params	[✓] typeof
[✓] catch	[✓] for	[✓] private	[✓] uint
[✓] char	[✓] foreach	[✓] protected	[✓] ulong
[✓] checked	[✓] goto	[✓] public	[✓] unchecked
[✓] class	[✓] if	[✓] readonly	[✓] unsafe
[✓] const	[✓] implicit	[✓] ref	[✓] ushort
[✓] continue	[✓] in	[✓] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[✓] virtual
[✓] default	[✓] interface	[✓] sealed	[ ] volatile
[✓] delegate	[✓] internal	[✓] short	[✓] void
[✓] do	[✓] is	[✓] sizeof	[✓] while
[✓] double	[ ] lock	[✓] stackalloc	
[✓] else	[✓] long	[✓] static	
[✓] enum	[✓] namespace	[✓] string	

3. unsafe code

- Útil para as seguintes situações...
  - *Interfaceamento* direto com o Sistema Operacional.
  - *Acesso direto* a dispositivos mapeados na memória...
    - ♦ Quando o acesso ao dispositivo é feito através de um intervalo de memória RAM.
      - ⇒ Exemplo: **RAM de vídeo**.
  - Implementação de algorítmos de **tempo crítico**.
- O que vem pela frente...
  - ***Ponteiros***.
  - ***Endereço de memória***.
  - Estilo de programação C++... porém utilizando C#.

4. safe x unsafe

- ***safe code***...
  - Value types.
  - Reference types.
- ***unsafe code***...
  - Value types.
  - Reference types.
  - **Pointer** types.

```
unsafe
{
    int i;    // value type
    object o; // reference type
    int* pi;  // pointer type
```

```
    void* pv; // pointer type (tipo desconhecido)
}
```

5. Exemplo

```
int i = 10;
unsafe
{
    int* p = &i;
    *p = 11;
    Console.WriteLine("Conteúdo: {0}", *p);
    Console.WriteLine("Endereço: {0:X}", (ulong)p);
}
Console.WriteLine("i = {0}", i); // 11
```

6. Tipos permitidos de ponteiros

- sbyte*, byte*
- short*, ushort*
- int*, uint*
- long*, ulong*
- char*
- float*, double*
- decimal*
- bool*
- void* (*um ponteiro para um tipo desconhecido*)
- Um ponteiro não pode "*apontar*" para um tipo referência ou para um tipo estrutura que contenha algum tipo referência.
- enumeração*
- int** (um ponteiro para um ponteiro)
- estrutura* (*desde que a estrutura contenha somente tipos valor*)
- int*[] (um array de ponteiros para o tipo int)

7. Por que o código é unsafe?

```
class Program
{
    static void Main(string[] args)
    {
        int[] numeros = new int[5] { 1, 2, 3, 4, 5 };
        unsafe
        {
            fixed (int* pNumero = &numeros[0])
            {
                for (int i = 0; i < 10; i++)
                {
                    Console.WriteLine(*(pNumero + i));
                    // *(pNumero + i) = pNumero[i]
                }
            }
        }
    }
}
```

8. Exemplo

```
public unsafe struct Node
{
    int value;
    Node* left;
    Node* right;

    public int Value
```

```
    {
        get
        {
            return value;
        }
        set
        {
            this.value = value;
        }
    }

    public Node* Left
    {
        get
        {
            return left;
        }
        set
        {
            left = value;
        }
    }

    public Node* Right
    {
        get
        {
            return right;
        }
        set
        {
            right = value;
        }
    }
}
```

9. Exemplo

```
public struct Node
{
    int value;
    unsafe Node* left;
    unsafe Node* right;

    public int Value
    {
        get
        {
            return value;
        }
        set
        {
            this.value = value;
        }
    }

    public unsafe Node* Left
    {
        get
        {
            return left;
        }
        set
        {
```

```
            left = value;
        }
    }

    public unsafe Node* Right
    {
        get
        {
            return right;
        }
        set
        {
            right = value;
        }
    }
}
```

10. Variáveis fixas e móveis

- Variáveis fixas são aquelas armazenadas em locais da memória que não sofrem a ação do Garbage Collector...
  - Variáveis locais do tipo valor.
  - O operador & (**address of**) permite obtermos o endereço de uma váriavel do tipo fixa.
- Variáveis móveis são aquelas que sofrem a ação do Garbage Collector...
  - Campos de classe.
  - O operador & (**address of**) só permite obtermos o endereço de um campo móvel dentro de um contexto fixo (**fixed**).

```
class Program
{
    static int i = 10;

    static void Main(string[] args)
    {
        int j = 0;
        unsafe
        {
            fixed (int* p1 = &i)
            {
                ...
            }
            int* p2 = &j;
        }
    }
}
```

11. Exemplo

```
static void Main(string[] args)
{
    double d = 123.456;
    unsafe
    {
        byte* p = (byte*)&d;
        for (int i = 0; i < sizeof(double); i++)
        {
            Console.Write("{0:X2} ", *p++);
        }
        Console.WriteLine();
        Console.ReadLine();
    }
}
```

12. Acesso a membro via ponteiro

```
public struct Point
{
    int x, y;

    public int X
    {
        get
        {
            return x;
        }
        set
        {
            x = value;
        }
    }

    public int Y
    {
        get
        {
            return y;
        }
        set
        {
            y = value;
        }
    }
}

static void Main(string[] args)
{
    Point point;
    unsafe
    {
        Point* p = &point;
        p->X = 1; // (*p).X = 1;
        p->Y = 2; // (*p).Y = 2;
    }
    Console.WriteLine(point.X);
    Console.WriteLine(point.Y);
}
```

13. Otimizando o desempenho com ponteiros

- Arrays na pilha de "*alto desempenho e baixo overhead*"... baixo overhead por não necessitar de alocação no heap.

```
// Sequência de Fibonacci
unsafe
{
    int* fibonacci = stackalloc int[25];
    int* p = fibonacci;
    *p++ = *p++ = 1;
    for (int i = 2; i < 25; i++, p++)
    {
        *p = p[-1] + p[-2];
    }
    for (int i = 0; i < 25; i++)
    {
        Console.WriteLine(fibonacci[i]);
    }
}
```

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368 e 75025.

14. C#

- Dúvidas?

1. Threads

2. Keywords...

[✓] abstract	[✓] event	[✓] new	[✓] struct
[✓] as	[✓] explicit	[✓] null	[✓] switch
[✓] base	[✓] extern	[✓] object	[✓] this
[✓] bool	[✓] false	[✓] operator	[✓] throw
[✓] break	[✓] finally	[✓] out	[✓] true
[✓] byte	[✓] fixed	[✓] override	[✓] try
[✓] case	[✓] float	[✓] params	[✓] typeof
[✓] catch	[✓] for	[✓] private	[✓] uint
[✓] char	[✓] foreach	[✓] protected	[✓] ulong
[✓] checked	[✓] goto	[✓] public	[✓] unchecked
[✓] class	[✓] if	[✓] readonly	[✓] unsafe
[✓] const	[✓] implicit	[✓] ref	[✓] ushort
[✓] continue	[✓] in	[✓] return	[✓] using
[✓] decimal	[✓] int	[✓] sbyte	[✓] virtual
[✓] default	[✓] interface	[✓] sealed	[✓] volatile
[✓] delegate	[✓] internal	[✓] short	[✓] void
[✓] do	[✓] is	[✓] sizeof	[✓] while
[✓] double	[✓] lock	[✓] stackalloc	
[✓] else	[✓] long	[✓] static	
[✓] enum	[✓] namespace	[✓] string	

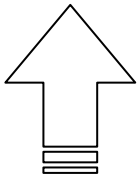
3. Thread

- *Vídeo pela Internet*...
  - **Thread 1** faz o *download* do vídeo a partir de um determinado site.
  - **Thread 2** *reproduz* o vídeo a medida que este vai sendo baixado.
  - As threads estão ***sincronizadas*** de tal forma que a reprodução (thread 2) só se inicia quando há um ***buffer suficiente*** de dados para manter a thread 2 em funcionamento... enquanto a thread 1 ainda continua com o download.

System.Threading.Thread => Controla as threads
System.Threading.Monitor => Utilizado na sincronização

4. Prioridade de execução

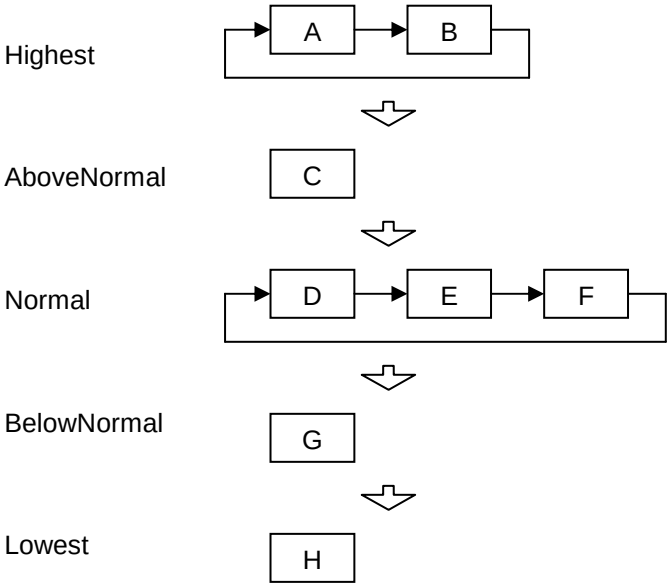
- ThreadPriority...



Highest
AboveNormal
Normal (*default*)
BelowNormal
Lowest

- Multitarefa...
  - As threads de igual prioridade compartilham uma ***fatia de tempo*** do processador (***time slice***).
  - Essa fatia de tempo é chamada de ***quantum***.
  - Após o término do quantum... a próxima thread de *igual prioridade* é executada.
  - A tarefa do escalanador de threads é manter as threads de prioridade mais alta executando o tempo todo.

5. Prioridade de execução



6. Prioridade de execução

```
static void Main(string[] args)
{
    Thread t1 = new Thread(new ThreadStart(T1));
    Thread t2 = new Thread(new ThreadStart(T2));
    t1.Priority = ThreadPriority.Lowest;
    t2.Priority = ThreadPriority.Highest;
    t1.Start();
    t2.Start();
    Console.WriteLine("Fim.");
    Console.ReadLine();
}

static void T1()
{
    for (int i = 0; i < 20; i++)
    {
        Console.WriteLine("T1 {0}: {1}", Thread.CurrentThread.Priority, i);
        for (int j = int.MinValue; j < int.MaxValue; j++) ;
    }
}

static void T2()
{
    for (int i = 0; i < 20; i++)
    {
        Console.WriteLine("T2 {0}: {1}", Thread.CurrentThread.Priority, i);
        for (int j = int.MinValue; j < int.MaxValue; j++) ;
    }
}
```

```
T2 Highest: 0
T2 Highest: 1
T1 Lowest: 0
Fim.
T2 Highest: 2
T2 Highest: 3
T2 Highest: 4
T2 Highest: 5
```

```
T2 Highest: 6
T2 Highest: 7
T2 Highest: 8
T2 Highest: 9
T2 Highest: 10
T2 Highest: 11
T2 Highest: 12
T2 Highest: 13
T2 Highest: 14
T2 Highest: 15
T2 Highest: 16
T2 Highest: 17
T2 Highest: 18
T2 Highest: 19
T1 Lowest: 1
T1 Lowest: 2
T1 Lowest: 3
T1 Lowest: 4
T1 Lowest: 5
T1 Lowest: 6
T1 Lowest: 7
T1 Lowest: 8
T1 Lowest: 9
T1 Lowest: 10
T1 Lowest: 11
T1 Lowest: 12
T1 Lowest: 13
T1 Lowest: 14
T1 Lowest: 15
T1 Lowest: 16
T1 Lowest: 17
T1 Lowest: 18
T1 Lowest: 19
```

7. Sincronização

```
class Numero
{
    decimal valor;

    public decimal Valor
    {
        get
        {
            return valor;
        }
    }

    public void Inc()
    {
        valor++;
    }
}
```

8. Sincronização (o problema)

```
class Program
{
    static Numero numero;

    static void Main(string[] args)
    {
        numero = new Numero();
```

```
        Thread t1 = new Thread(new ThreadStart(T1));
        Thread t2 = new Thread(new ThreadStart(T2));
        t1.Start();
        t2.Start();
        Console.ReadLine();
        Console.WriteLine(numero.Valor);
    }

    static void T1()
    {
        for (int i = 0; i < 1000000; i++)
        {
            numero.Inc();
        }
        Console.WriteLine("T1 terminou.");
    }

    static void T2()
    {
        for (int i = 0; i < 1000000; i++)
        {
            numero.Inc();
        }
        Console.WriteLine("T2 terminou.");
    }
}
```

9. Sincronização (Monitor)

```
class Numero
{
    decimal valor;

    public decimal Valor
    {
        get
        {
            return valor;
        }
    }

    public void Inc()
    {
        Monitor.Enter(this);
        valor++;
        Monitor.Exit(this);
    }
}
```

10. Sincronização (lock)

```
class Numero
{
    decimal valor;

    public decimal Valor
    {
        get
        {
            return valor;
        }
    }

    public void Inc()
```

```
    {
        lock (this)
        {
            valor++;
        }
    }
}
```

11. Join

```
class Program
{
    static int result;

    static void Main(string[] args)
    {
        result = 0;
        Thread t = new Thread(new ThreadStart(Thread2));
        t.Start();
        t.Join(); // espera até que "t" termine.
        Console.WriteLine(result);
    }

    static void Thread2()
    {
        result = 123;
    }
}
```

12. volatile

```
class Program
{
    static int result;
    static volatile bool finished;

    static void Main(string[] args)
    {
        result = 0;
        finished = false;
        Thread t = new Thread(new ThreadStart(Thread2));
        t.Start();
        while (!finished) ;
        Console.WriteLine(result);
    }

    static void Thread2()
    {
        result = 123; // o compilador, runtime, hardware
        finished = true; // pode inverter estas instruções para otimizar
    }
}
```

13. Deadlock

Thread 1

```
lock (a)
{
    ...
    lock (b)
    {
        ...
    }
```

```
}
```

Thread 2

```
lock (b)
{
    ...
    lock (a)
    {
        ...
    }
}
```

14. C#

- Dúvidas?

1. C# 2.0

- Extensões da linguagem...
  - Generics.
  - Anonymous methods.
  - Iterators.
  - Partial types.

2. StoreInt

```
public class StoreInt
{
    int[] items;

    public StoreInt(int count)
    {
        items = new int[count];
    }

    public int this[int index]
    {
        get
        {
            return items[index];
        }
        set
        {
            items[index] = value;
        }
    }
}

StoreInt store = new StoreInt(3);
store[0] = 10;
store[1] = 20;
store[2] = 30;
Console.WriteLine(store[0]);
Console.WriteLine(store[1]);
Console.WriteLine(store[2]);
```

3. StoreString

```
public class StoreString
{
    string[] items;

    public StoreString(int count)
    {
        items = new string[count];
    }

    public string this[int index]
    {
        get
        {
            return items[index];
        }
        set
        {
            items[index] = value;
        }
    }
}
```

```
StoreString store = new StoreString(3);
store[0] = "a";
store[1] = "b";
store[2] = "c";
Console.WriteLine(store[0]);
Console.WriteLine(store[1]);
Console.WriteLine(store[2]);
```

4. StoreInt x StoreString

```
public class StoreInt
{
    int[] items;

    public StoreInt(int count)
    {
        items = new int[count];
    }

    public int this[int index]
    {
        get
        {
            return items[index];
        }
        set
        {
            items[index] = value;
        }
    }
}

public class StoreString
{
    string[] items;

    public StoreString(int count)
    {
        items = new string[count];
    }

    public string this[int index]
    {
        get
        {
            return items[index];
        }
        set
        {
            items[index] = value;
        }
    }
}
```

5. StoreAll

```
public class StoreAll
{
    object[] items;

    public StoreAll(int count)
    {
        items = new object[count];
```

```
    }

    public object this[int index]
    {
        get
        {
            return items[index];
        }
        set
        {
            items[index] = value;
        }
    }
}
```

6. StoreAll (Problemas)

```
// StoreAll será utilizada com o propósito
// de armazenar números.
StoreAll store = new StoreAll(3);
store[0] = 10;
store[1] = "b"; // compile-time type checking (impossível)
store[2] = 30;
Console.WriteLine(
    (int)store[0] + // casting dos dados é requerido
    (int)store[2]   // performance penalty
);
```

7. StoreAll (versão utilizando generics)

```
public class StoreAll<T>
{
    T[] items;

    public StoreAll(int count)
    {
        items = new T[count];
    }

    public T this[int index]
    {
        get
        {
            return items[index];
        }
        set
        {
            items[index] = value;
        }
    }
}

// StoreAll será utilizada com o propósito
// de armazenar números.
StoreAll<int> store = new StoreAll<int>(3);
store[0] = 10;
store[1] = "b"; // erro: compile-time type checking
store[2] = 30;
// casting dos dados não é requerido.
// não há perda de performance.
Console.WriteLine(
    store[0] +
    store[2]
);
```

8. Generics

- Classes, estruturas, interfaces, delegates e métodos ***podem ser parametrizados*** por tipos de dados que armazenam e/ou manipulam.
  - Vantagens...
    - compile-time type checking.
    - runtime casting desnecessário.

```
StoreAll<int> store = new StoreAll<int>(3);
store[0] = 10;
store[1] = "b"; // erro: compile-time type checking
store[2] = 30;
// casting dos dados não é requerido.
// não há perda de performance.
Console.WriteLine(
    store[0] +
    store[2]
);
```

9. Generics

- C# Generics é semelhante ao conceito de generics em...
  - Eiffel.
  - Ada.
  - C++ templates (mas ***sem as complicações*** dos templates em C++).

10. Estrutura de dados (object)

```
public class Stack
{
    object[] items;
    int count;

    public void Push(object item)
    {
        ...
    }

    public object Pop()
    {
        ...
    }
}

Stack stack = new Stack();
stack.Push("abc");
stack.Push(1);
int number = (int)stack.Pop();
```

11. Estrutura de dados (Generics)

```
public class Stack<T>
{
    T[] items;
    int count;

    public void Push(T item)
    {
        ...
    }

    public T Pop()
```

```
    {
        ...
    }
}

Stack<int> stack = new Stack<int>();
stack.Push(1);
int number = stack.Pop();
```

12. Generics

```
public class Dictionary<K, V>
{
    public void Add(K key, V value)
    {
        ...
    }

    public V this[K key]
    {
        get { ... }
        set { ... }
    }
}

Dictionary<int, string> dic = new Dictionary<int, string>();
dic.Add(1, "um");
dic.Add(2, "dois");
string numero = dic[1];
```

13. Generics

```
public class Dictionary<K, V>
{
    public void Add(K key, V value)
    {
        if (key.CompareTo(...) < 0) // CompareTo????
        {
        }
    }
}

public class Dictionary<K, V> where K : IComparable
{
    public void Add(K key, V value)
    {
        if (key.CompareTo(...) < 0)
        {
        }
    }
}
```

14. Generics

```
public class EntityTable<K, E>
    where K : IComparable
    where E : Entity, new()
{
    public void Add(K key, E entity)
    {
        if (key.CompareTo(...))
        {
        }
    }
```

```
}
```

15. Generic Methods

```
public static void PushAll(Stack<int> stack, params int[] values)
{
    foreach (int value in values)
    {
        stack.Push(value);
    }
}

public static void PushAll<T>(Stack<T> stack, params T[] values)
{
    foreach (T value in values)
    {
        stack.Push(value);
    }
}

Stack<string> stack1 = new Stack<string>();
Stack<int> stack2 = new Stack<int>();
PushAll<string>(stack1, "a", "b");
PushAll<int>(stack2, 1, 2);
```

16. Anonymous methods

```
public class Pontuacao : object
{
    int pontos;

    public Pontuacao(int pontos) : base()
    {
        this.pontos = pontos;
    }

    public event EventHandler Zero;

    public void AddPontos(int pontos)
    {
        this.pontos += pontos;
        if (this.pontos == 0)
        {
            OnZero();
        }
    }

    protected virtual void OnZero()
    {
        if (Zero != null)
        {
            Zero(this, EventArgs.Empty);
        }
    }
}
```

17. Anonymous methods

```
class Program
{
    static void Main(string[] args)
    {
        Pontuacao p = new Pontuacao(5);
        p.Zero += new EventHandler(p_Zero);
```

```
        p.AddPontos(2);
        p.AddPontos(-7);
    }

    static void p_Zero(object sender, EventArgs e)
    {
        Console.WriteLine("Zero pontos.");
    }
}
```

18. Anonymous methods

- Permite que o código associado a um evento seja escrito **in-line**.
- C# 2.0 suporta um recurso conhecido como "***closures***".

```
class Program
{
    static void Main(string[] args)
    {
        Pontuacao p = new Pontuacao(5);
        p.Zero += delegate(object sender, EventArgs e)
        {
            Console.WriteLine("Zero pontos.");
            p.AddPontos(1); // Closures
            // p -> outer variable
            // tempo de vida de p?
        };
        p.AddPontos(2);
        p.AddPontos(-7);
    }
}
```

19. Numbers

```
Numbers numbers = new Numbers(5, 3, 8, 2, 0, -1, 9, -2);
foreach (int number in numbers)
{
    Console.WriteLine(number);
}
```

20. Numbers => IEnumerable

```
public class Numbers : object, IEnumerable
{
    int[] numbers;

    public Numbers(params int[] numbers)
    {
        this.numbers = numbers;
    }

    public int this[int index]
    {
        get
        {
            return numbers[index];
        }
        set
        {
            numbers[index] = value;
        }
    }

    public int Length
```

```
    {
        get
        {
            return numbers.Length;
        }
    }

    public IEnumerator GetEnumerator()
    {
        return new NumbersEnumerator(this);
    }
}
```

21. Numbers => IEnumerator

```
public class NumbersEnumerator : object, IEnumerator
{
    Numbers numbers;
    int pos;

    public NumbersEnumerator(Numbers numbers)
    {
        this.numbers = numbers;
        this.pos = -1;
    }

    public object Current
    {
        get
        {
            if (pos > -1 && pos < numbers.Length)
            {
                return numbers[pos];
            }
            else
            {
                throw new InvalidOperationException();
            }
        }
    }

    public bool MoveNext()
    {
        if (pos >= numbers.Length)
        {
            return false;
        }
        else
        {
            pos++;
            return pos < numbers.Length;
        }
    }

    public void Reset()
    {
        pos = -1;
    }
}
```

22. Iterator

- Um iterador é um bloco que retorna "**yield return**" uma sequência de valores.

```
public class Numbers : object, IEnumerable
{
    int[] numbers;

    public Numbers(params int[] numbers)
    {
        this.numbers = numbers;
    }

    public IEnumerator GetEnumerator()
    {
        foreach (int number in numbers)
        {
            yield return number;
        }
    }
}
```

23. Partial types

- Classes, estruturas e interfaces podem ser quebradas em diferentes unidades de compilação.
- Aplicação...
  - *Machine-generated* code (geradores de código fonte) separado da implementação escrita pelo programador.

```
public partial class Cliente : object
{
    int id;
    string nome;
    string fone;
}

public partial class Cliente : object
{
    public void DoSomething()
    {
    }
}
```

1. Estruturas de Dados

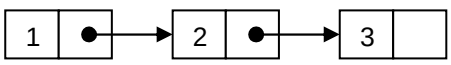
- *Estáticas*...
  - Possuem **tamanho fixo**. Ex. Arrays.
- *Dinâmicas*...
  - **Crescem e diminuem** em tempo de execução.
    - **Listas encadeadas**... coleções de itens de dados organizados em uma linha, inserções e exclusões são permitidas em qualquer parte da lista.
    - **Pilhas**... importantes em compiladores e sistemas operacionais... as inserções e exclusões são sempre feitas na extremidade da pilha.
    - **Filas**... representam linhas de espera... as inserções são feitas no final e as exclusões são feitas a partir do início.

2. Classes auto-referenciais

```
public class Node
{
    int numero;
    Node next;

    public Node(int numero)
    {
        this.numero = numero;
    }

    public int Numero
    {
        get
        {
            return numero;
        }
    }

    public Node Next
    {
        get
        {
            return next;
        }
        set
        {
            next = value;
        }
    }
}
```

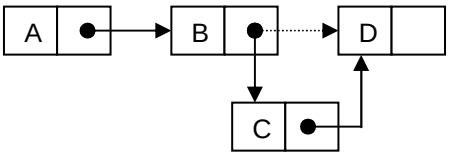
3. Classes auto-referenciais

```
Node n1 = new Node(1);
Node n2 = new Node(2);
Node n3 = new Node(3);
n1.Next = n2;
n2.Next = n3;
n3.Next = null;
Node current = n1;
while (current != null)
{
    Console.WriteLine(current.Numero);
    current = current.Next;
}
```



4. Listas encadeadas

- Coleção ***linear*** de objetos de classe auto-referencial (chamados ***nós***)... conectados por encadeamento de referência... daí o termo lista encadeada.
- As **pilhas** e **filas**, que também são estruturas de dados ***lineares***, são versões restritas das listas encadeadas.



5. Arrays x Listas encadeadas

- Os arrays possuem **tamanho fixo**... e as listas encadeadas possuem **tamanho variável**.
- Um array pode ser declarado para conter mais elementos do que o previsto a custo de um **desperdício de memória**... as listas encadeadas fornecem uma **melhor utilização da memória**, e permitem que o programa se adapte em tempo de execução.
- Os elementos de um array são armazenados de maneira contígua na memória... o endereço de qualquer elemento pode ser **calculado diretamente** a partir de seu deslocamento em relação ao início... já nas listas encadeadas, tal procedimento só é possível **percorrendo-se a lista desde o início**.
- Inserir um elemento em um array ordenado requer que todos os elementos subsequentes sejam **movidos**... numa lista encadeada ordenada, após localizar o ponto de inserção para um novo item, **inserir um elemento é rápido**... apenas duas referências precisam ser modificadas, e todos os **nós** existentes permanecem em suas posições na memória.

6. Implementação .net de uma lista encadeada

- A classe **LinkedList<T>** implementa uma lista encadeada ***duplamente ligada*** (**nó** anterior e **nó** seguinte).

```
LinkedList<int> list = new LinkedList<int>();
list.AddLast(1);
list.AddLast(2);
list.AddLast(3);
foreach (int numero in list)
{
    Console.WriteLine(numero);
}
```

7. Pilhas

- Uma pilha é uma ***versão restrita*** de uma **lista encadeada**.
- A pilha recebe e libera **nós** *apenas no início*.
- Estrutura de dados **LIFO** (Last-In First-Out).

```
Stack<int> stack = new Stack<int>();
stack.Push(1);
stack.Push(2);
stack.Push(3);
while (stack.Count > 0)
{
    Console.WriteLine(stack.Pop());
}
```

8. Filas

- Uma fila também é uma ***versão restrita*** de uma **lista encadeada**.
- A fila recebe **nós** no final e libera **nós** *no início*.
- Estrutura de dados **FIFO** (First-In First-Out).

```
Queue<int> queue = new Queue<int>();
queue.Enqueue(1);
queue.Enqueue(2);
queue.Enqueue(3);
while (queue.Count > 0)
{
    Console.WriteLine(queue.Dequeue());
}
```

9. Hashtables

- Vamos armazenar 100 clientes na memória (utilizando-se arrays)...
- O acesso será feito através de seu código de 6 dígitos (1 a 999.999)...
- Precisaremos então de um array de 999.999 elementos...
- A inserção e consulta de registros oferece um desempenho elevadívissimo... mas o desperdício de memória é alto...

10. Hashtables

- Uma solução melhor seria converter cada código de cliente em um índice exclusivo no array... desta forma obteríamos a performance requerida sem o desperdício exagerado de memória...
- Este esquema de conversão é a base de uma técnica chamada **hashing**...
- Por quê hashing...
  - Quando convertemos uma chave em um índice de array, literalmente misturamos os bits, formando um tipo de bagunça de números (significado de hash).

11. Hashtables

- Há entretanto um problema...
- Podem haver colisões, ou seja, duas chaves podem, através do processo de hashing, produzir o mesmo índice no array...
- Soluções...
  - Fazer um novo hash sobre o hash gerado...
  - Pesquisar sequencialmente a primeira célula vazia caso a célula atual esteja ocupada...
  - Tornar cada célula no array como sendo na verdade um recipiente de hash... normalmente uma lista encadeada de todos os pares chave/valor que se encontram nessa célula. Essa é a solução adotada pela classe **Hashtable** da plataforma **.net**.

12. Hashtables

- Fator de carga...
  - Afeta o desempenho das tabelas hash...
  - É a proporção do número de células ocupadas com relação a quantidade delas...
  - Quanto mais a proporção se aproxima de 1... maiores as chances de colisões...
  - Aumentando-se o fator de carga... obtemos melhor utilização da memória... mas o programa funciona mais lentamente...
  - Diminuindo-se o fator de carga... o programa funciona mais rapidamente... mas a custo de uma maior utilização da memória...

13. Hashtables

```
Hashtable n1 = new Hashtable();
n1.Add(1, "um");
n1.Add(2, "dois");
n1.Add(3, "três");
n1.Add(4, "quatro");
n1.Add(5, "cinco");
Console.WriteLine(n1[3]);

// Dictionary é um hash otimizado
Dictionary<int, string> n2 = new Dictionary<int, string>();
n2.Add(1, "um");
```

```
n2.Add(2, "dois");
n2.Add(3, "três");
n2.Add(4, "quatro");
n2.Add(5, "cinco");
Console.WriteLine(n2[3]);
```

IDisposable...

```
public class ResourceX : IDisposable
{
    #region Fields

    #pragma warning disable 0414

    string managedResource1;
    int managedResource2;
    bool disposed = false;

    #pragma warning restore 0414

    #endregion

    #region Constructor

    public ResourceX()
    {
        managedResource1 = "some string";
        managedResource2 = 1;
        // alocação de recurso não gerenciável vem aqui (resource X)
    }

    #endregion

    #region Destructor

    ~ResourceX()
    {
        Dispose(false);
    }

    #endregion

    #region Disposing

    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // dispose managed resources
                // se existe uma instância de uma classe (membro)
                // que também implementa
                // IDisposable, então: instanceMember.Dispose()
            }
            // recursos não gerenciáveis são liberados aqui
            disposed = true;
        }
    }

    #endregion

    #region IDisposable Members

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
```

```
    #endregion
}

public class ResourceXandY : ResourceX
{
    #region Fields

    #pragma warning disable 0414

    string managedResource;
    bool disposed = false;

    #pragma warning restore 0414

    #endregion

    #region Constructor

    public ResourceXandY() : base()
    {
        managedResource = "another string";
        // alocação de recurso não gerenciável vem aqui (resource Y)
    }

    #endregion

    #region Disposing

    protected override void Dispose(bool disposing)
    {
        if (!disposed)
        {
            try
            {
                if (disposing)
                {
                    // dispose recurso não gerenciável nesta classe
                    // se existe uma instância de uma classe (membro)
                    // que também implementa
                    // IDisposable, então: instanceMember.Dispose()
                }
                // recursos não gerenciáveis são liberados aqui
                disposed = true;
            }
            finally
            {
                base.Dispose(disposing);
            }
        }
    }

    #endregion
}

using (ResourceXandY a = new ResourceXandY())
// ResourceX constructor invoked => unmanaged resource X allocated
// ResourceXandY constructor invoked => unmanaged resource Y allocated
{
   ...
} // Uma chamada a Dispose é colocada aqui (implicitamente) pelo compilador C#
  // Dispose chama o método virtual Dispose(bool)
  // Para o caso em que o método Dispose() não é chamado, ResourceX destructor
  // chamará o método Dispose(bool) para liberar recursos não gerenciáveis
```

IComparable...

```
public class NivelGrau : IComparable<NivelGrau>
{
    #region Fields

    int nivel, grau;

    #endregion

    #region Constructor

    public NivelGrau(int nivel, int grau)
    {
        this.nivel = nivel;
        this.grau = grau;
    }

    #endregion

    #region IComparable<NivelGrau> Members

    public int CompareTo(NivelGrau other)
    {
        if (this.nivel == other.nivel && this.grau == other.grau)
        {
            return 0;
        }
        else
        {
            if (this.nivel < other.nivel ||
               (this.nivel == other.nivel && this.grau < other.grau))
            {
                return -1;
            }
            else
            {
                return 1;
            }
        }
    }

    #endregion
}
```

Threads... (bom design)

```
class Calling
{
    public void Call()
    {
        Thread t1 = new Thread(new ThreadStart(Working.Worker1));
        t1.Name = "ThreadName1";
        Thread t2 = new Thread(new ThreadStart(Working.Worker2));
        t2.Name = "ThreadName2";
        t2.Priority = ThreadPriority.BelowNormal;
        t2.IsBackground = true;
        // o processo terminará independente de t2 estiver ativa ou não
        t1.Start();
        t2.Start();
    }
}

public class Working
```

```
{
    #region Fields

    // task 1
    // task 2

    #endregion

    #region Worker 1 (Processor 1)

    public static void Worker1()
    {
        // working on task 1
    }

    #endregion

    #region Worker 2 (Processor 2 or sharing time with processor 1)

    public static void Worker2()
    {
        // working on task 2
    }

    #endregion
}
```

Gerenciamento de memória... (string x StringBuilder)

```
string alfabeto = "";
for (int i = 'A'; i <= 'Z'; i++)
{
    alfabeto += (char)i;
}
Console.WriteLine(alfabeto);

StringBuilder alfabeto = new StringBuilder();
for (int i = 'A'; i <= 'Z'; i++)
{
    alfabeto.Append((char)i);
}
Console.WriteLine(alfabeto.ToString());
```

Recomendações de leitura...

- Livros...
  - Iniciante/Intermediário...
    - Professional C# Programando – Simon Robinson (***Excelente***): Makron Books.
    - C# Como Programar – Deitel (***Excelente explicação de orientação a objeto***): Makron Books.
    - Beginning C# Programando – Karli Watson: Makron Books.
  - Avançado...
    - The C# Programming Language – Anders Hejlsberg / Scott Wiltamuth / Peter Golde (Inglês): Addison Wesley.
- Documentação online...
  - SDK.

Obrigado, e boa sorte a todos!!!

Fábio.