

Ambientes Virtuais de Execução

Estrutura de Tipos

Instâncias de Tipos

- ▶ Uma instância de um tipo é um objecto ou um valor.
- ▶ Um **objecto** é uma instância de um tipo num *garbage collected heap*.
- ▶ As instâncias de um **tipo valor** não são objectos:
 - ▶ Não têm cabeçalho do objecto.
 - ▶ Não estão alocados como entidades distintas no *garbage collected heap*.

Membros dos Tipos

- ▶ Os tipos podem conter membros:
 - ▶ De instância
 - ▶ De tipo (*static*)
- ▶ Os membros de um tipo podem ser:
 - ▶ Constantes (membros estáticos)
 - ▶ Campos (*Fields*)
 - ▶ Métodos
 - ▶ Construtores
 - ▶ de instância
 - ▶ de tipo
 - ▶ Propriedades
 - ▶ *Nested Types* (são sempre membros *static*)
 - ▶ Eventos

Diferentes tipos de membros - Exemplo

```
public sealed class SomeType {  
    // Nested class  
    private class SomeNestedType { }  
    // Constant, read-only, and static read/write field  
    private const    Int32  c_SomeConstant        = 1;  
    private readonly String m_SomeReadOnlyField    = "2";  
    private static   Int32  s_SomeReadWriteField = 3;  
  
    // Type constructor  
    static SomeType() { }  
    // Instance constructors  
    public SomeType() { }  
    public SomeType(Int32 x) { }  
    //....  
}
```

Diferentes tipos de membros - Continuação

// Static and instance methods

```
public static void Main() { }
```

```
public String InstanceMethod() { return null; }
```

// Instance property

```
public Int32 SomeProp {  
    get { return 0; }  
    set { }  
}
```

// Instance parameterful property (indexer)

```
public Int32 this[String s] {  
    get { return 0; }  
    set { }  
}
```

// Instance event

```
public event EventHandler SomeEvent;  
}
```

Outros Membros dos Tipos

- ▶ Existem outros membros que não fazem parte do CLS
 - ▶ Sobrecarga de operadores
 - ▶ Operadores de conversão

Visibilidade de um tipo

IL Term	C#Term	Visual Basic Term	Descrição
private	internal (por omissão)	Friend	Visível apenas dentro do <i>assembly</i>
public	public	Public	Visível dentro e fora do <i>assembly</i>

Acessibilidade dos membros de um tipo

IL Term	C# Term	Visual Basic Term	Descrição
Private	private (default)	Private	Acessível apenas pelos métodos do tipo
Family	protected	Protected	Acessível apenas pelos métodos do tipo e dele derivados, dentro ou fora do assembly
Family and Assembly	(não suportada)	(não suportada)	Acessível apenas pelos métodos do tipo e dele derivados dentro do assembly
assembly	internal	Friend	Acessível apenas pelos métodos de tipos definidos no assembly
Family or Assembly	protected internal	Protected Friend	Acessível apenas pelos métodos do tipo e dele derivados dentro ou fora do assembly e pelos métodos de outros tipos do assembly
public	public	Public	Acessível por métodos de qualquer tipo



Acessibilidade de um membro

- ▶ Para qualquer membro ser acessível, tem de estar definido num tipo que seja visível.
- ▶ Ao derivar de uma classe base:
 - ▶ CLR permite que a acessibilidade de um membro redefinido (*overriden*) fique **menos restritiva**.
 - ▶ Em C# é necessário que a acessibilidade **seja a mesma**.

Herança

- ▶ Um tipo **não pode ter** mais que uma **classe base**.
- ▶ Um tipo pode implementar **qualquer número de interfaces**.

IL Term	C# Term	Visual Basic Term	Description
abstract	abstract	MustInherit	Tipo abstracto
final	sealed	NotInheritable	Não pode ser estendido.

- ▶ Apenas um dos modificadores pode ser aplicado a um tipo.

Classes estáticas (C#)

- ▶ Não podem ser instanciadas;
 - ▶ O compilador não produz nenhum construtor de instância.
- ▶ Não podem implementar interfaces.
- ▶ Só podem definir membros **estáticos**.
- ▶ O compilador de C# torna este tipo de classes *abstract* e *sealed*.

Atributos pré definidos aplicáveis a métodos (1)

CLR Term	C# Term	Visual Basic Term	Descrição
Static	static	Shared	Método associado ao próprio tipo, não a uma instância do tipo. Os membros estáticos não podem aceder a campos de instância ou métodos de instância.
Instance	(default)	(default)	Método associado a uma instância do tipo. Pode aceder aos campos e métodos de instância, assim como aos campos e métodos estáticos.
Virtual	virtual	Overridable	O método mais derivado é invocado mesmo que o objecto seja convertido para um tipo base. Aplica-se apenas a métodos de instância.

Atributos pré-definidos aplicáveis a métodos (2)

CLR Term	C# Term	Visual Basic Term	Descrição
NewSlot	new (default)	Shadows	O método não deve redefinir um método virtual definido pelo seu tipo base; o método esconde o método herdado . NewSlot aplica-se apenas a métodos virtuais.
Override	override	Overrides	Indica que o método está a redefinir um método virtual definido pelo seu tipo base. Aplica-se apenas a métodos virtuais .
Abstract	abstract	MustOverride	Indica que um tipo derivado tem de implementar um método com uma assinatura que corresponda a este método abstracto. Um tipo com um método abstracto é um tipo abstracto. Aplica-se apenas a métodos virtuais .
Final	sealed	NotOverridable	Um tipo derivado não pode redefinir este método . Aplica-se apenas ao métodos virtuais .

Invocação de métodos em IL - call

- ▶ A instrução **call** é utilizada para invocar métodos estáticos, de instância e virtuais
 - ▶ Quando é usado para **invocar um método estático**, tem que se especificar o **tipo** que define o método que o CLR vai invocar.
 - ▶ Quando é usado para **invocar um método de instância ou virtual**, é necessário especificar uma variável que se refere ao objecto.
 - ▶ o **tipo da variável** indica que tipo define o método que o CLR deve invocar
 - ▶ se o tipo da variável não define o método, os tipos base são verificados para um método correspondente.

Invocação de métodos em IL - callvirt

- ▶ A instrução **callvirt** pode ser utilizada para invocar métodos de instância ou virtuais.
 - ▶ Quando é usado para **invocar um método de instância virtual**, o CLR descobre o tipo actual do objecto que está a invocar o método.
 - ▶ **Invoca o método polimorficamente.**
 - ▶ Quando é usado **para invocar um método de instância não virtual**, o tipo da variável indica o tipo que define o método que o CLR vai invocar.
 - ▶ Se a variável que referencia o objecto for null, lança excepção do tipo `NullReferenceException`.

Exemplo 1

```
using System;

public class TesteA{
    public static void Main(){
        Console.WriteLine( );
        Object o = new Object();
        o.GetHashCode();
    }
}
```

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size          15 (0xf)
    .maxstack 1
    .locals init (object V_0)
    IL_0000: nop
    IL_0001: call void [mscorlib]System.Console::WriteLine()
    IL_0006: nop
    IL_0007: newobj instance void [mscorlib]System.Object::.ctor()
    IL_000c: stloc.0
    IL_000d: ldloc.0
    IL_000e: callvirt instance int32
                                   [mscorlib]System.Object::GetHashCode()
    IL_0013: pop
    IL_0014: ret
} // end of method TesteA::Main
```



Exemplo 1

```
using System;

public class TesteA{
    public static void Main(){
        Object o = new Object();
        o.GetType();
    }
}
```

```
.method public hidebysig static void Main() cil managed
{
    .entrypoint
    // Code size          15 (0xf)
    .maxstack 1
    .locals init (object V_0)
    IL_0000:  nop
    IL_0001:  newobj instance void [mscorlib]System.Object::.ctor()
    IL_0006:  stloc.0
    IL_0007:  ldloc.0
    IL_0008:  callvirt instance class [mscorlib]System.Type
               [mscorlib]System.Object::GetType()
    IL_000d:  pop
    IL_000e:  ret
} // end of method TesteA::Main
```

Métodos de instância não virtuais em C#

- ▶ Em C#, nas chamadas aos **métodos de instância não virtuais** para tipos referência é utilizada a instrução **callvirt**
 - ▶ verifica que o objecto que está a fazer esta invocação não é null. Se for lança excepção do tipo `NullReferenceException`
- ▶ Ex:

```
using System;
public sealed class Program{
    public Int32 GetFive() { return 5;}
    public static void Main( ){
        Program p = null;
        Int32 x = p.GetFive();
    }
```




Métodos virtuais – instrução call

- ▶ Por vezes o compilador usa a instrução **call** para invocar métodos virtuais

- ▶ Ex:

```
internal class SomeClass{  
    public override String toString(){  
        return base.toString()  
    }  
}
```



Exemplos

```
using System;
class A{
    public void F(){ Console.WriteLine("A.F"); }
    public virtual void G(){ Console.WriteLine( " A.G");}
}
class B: A{
    public void F(){ Console.WriteLine("B.F");}
    public override void G() { Console.WriteLine("B.G");}
}
class Test{
    static void Main(){
        B b = new B(); A a = new B();
        a.F(); b.F(); a.G(); b.G();
    }
}
```



Exemplos

```
using System;
class A{
    public void F(){ Console.WriteLine("A.F"); }
    public virtual void G(){ Console.WriteLine( " A.G");}
}
class B: A{
    public void F(){ Console.WriteLine("B.F");}
    public override void G() { Console.WriteLine("B.G");}
}
class Test{
    static void Main(){
        B b = new B(); A a = new B();
        a.F(); b.F(); a.G(); b.G();
    }
}
```



A.F
B.F
B.G
B.G

Métodos com número variável de argumentos

```
class TParams {  
    static void showArgs(params object[] args) {  
        foreach(object arg in args)  
            Console.WriteLine(arg.ToString());  
    }  
  
    static void Main() {  
        showArgs("olá", "admirável", "mundo", "novo!");  
    }  
}
```

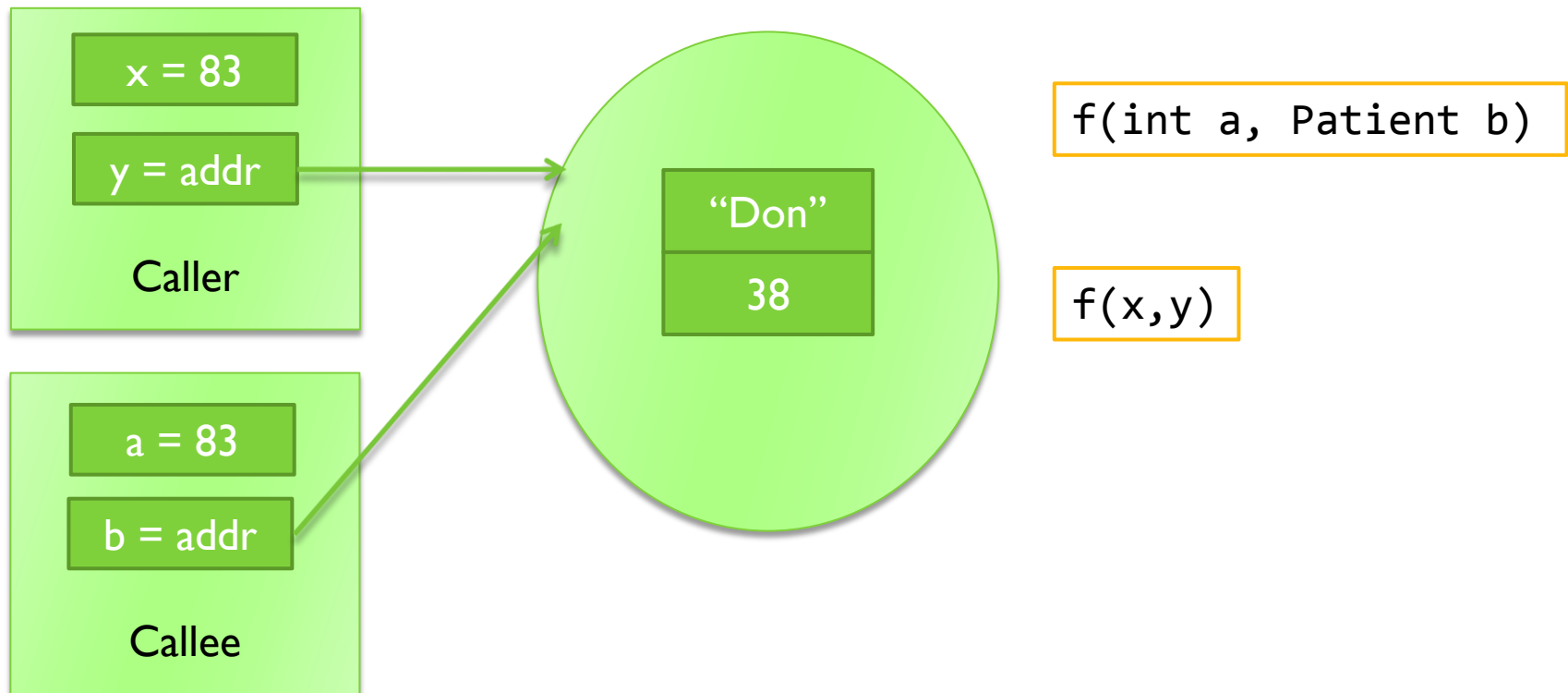
Apenas o último parâmetro do método pode ser um *parameter array*.



Passagem de parâmetros: por valor

► Passagem por valor:

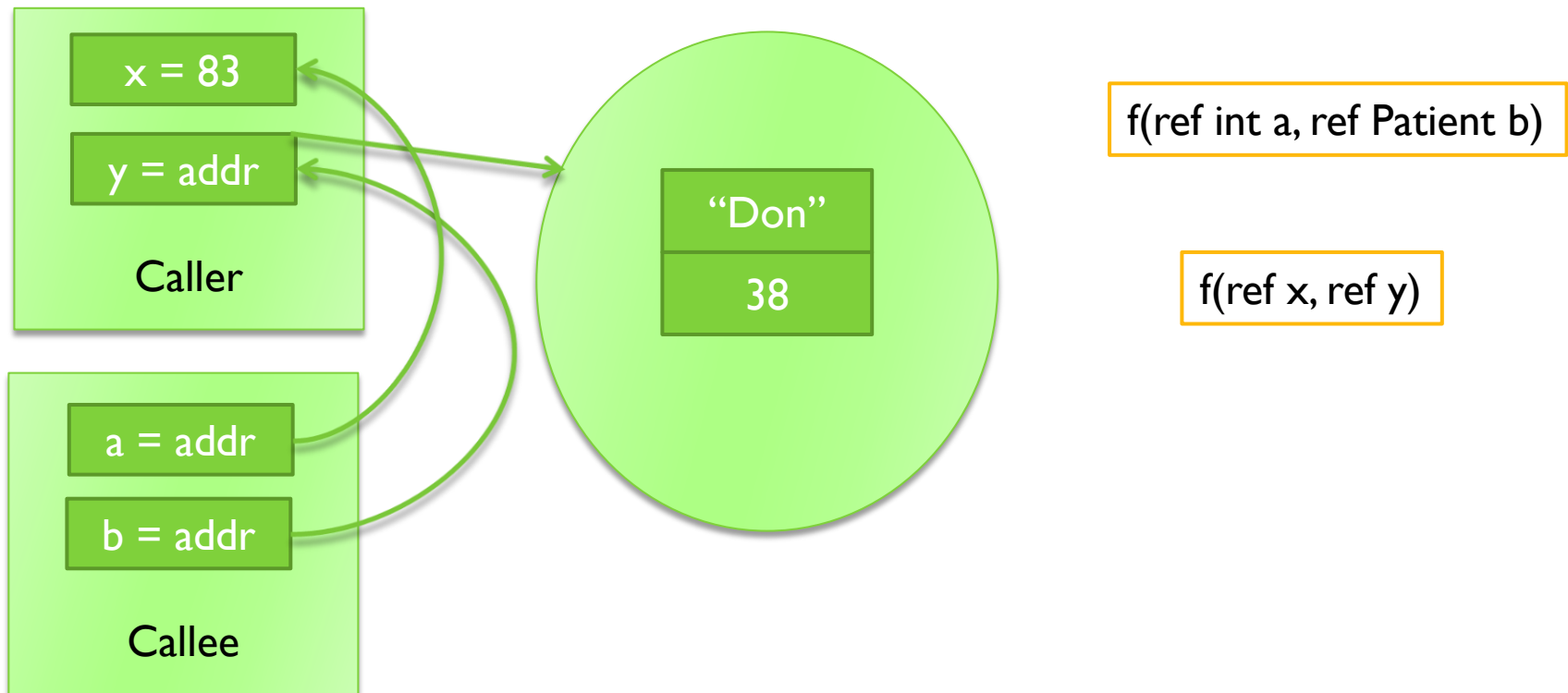
- Através da cópia do conteúdo da variável;
- Comportamento por omissão.



Passagem de parâmetros: por referência

► Passagem por referência:

- Através de ponteiro *managed* para a variável;
- Em IL indicado por `&` e em C# por `ref`.



ref versus out

```
using System;
public sealed class Program{
    public static void Main(){
        Int32 x;
        GetVal( out x );
        Console.WriteLine(x);
    }
    private static
        void GetVal(out Int32 v){
            v = 10;
        }
}
```

GetVal tem de inicializar v

```
using System;
public sealed class Program{
    public static void Main(){
        Int32 x = 5;
        GetVal( ref x );
        Console.WriteLine(x);
    }
    private static
        void GetVal(ref Int32 v){
            v = 10;
        }
}
```

x tem de ser inicializada.



ref versus out

//Pseudo - código

```
using System;
Using System.IO;
public sealed class Program{
    public static void Main(){
        FileStream fs;
        StartProcessingFiles( out fs );
        for( ; fs !=null; ContinueProcessingFiles( ref fs ) ){ fs.Reader( . . . ); }
    }
    private static void StartProcessingFiles(out FileStream fs ){
        fs = new FileStream( ... );
    }
    private static void ContinueProcessing( ref FileStream fs ){
        fs.Close();
        if(noMoreFileToProcess) fs = null;
        else fs = new FileStream( ... )
    }
}
```



Parâmetros ref e a compatibilidade de tipos

```
class Utils {  
    public static void swap(ref object a, ref object b) {  
        object aux=a;  
        a=b;  
        b=aux;  
    }  
}
```

Sejam duas referências para string:
Qual o problema do seguinte código?

```
string s1,s2;  
Utils.swap(ref s1, ref s2)
```

Construtores

► CLR suporta:

► Construtores de tipo

- Utilizados para estabelecer o **estado inicial de um tipo**.
- É um método estático com um nome especial → **.cctor**.
- O CLR garante a chamada ao construtor de tipo antes de qualquer acesso a um campo de tipo.

► Construtores de instância

- Utilizados para estabelecer o **estado inicial de uma instância de um tipo**.
- É um método estático com um nome especial → **.ctor**.

Iniciação de tipos

- ▶ Em CLR, um construtor de tipo pode ser aplicado a interfaces, tipos referência e tipos valor
 - ▶ Não é permitido em C# aplicar construtores de tipos a interfaces.
 - ▶ Por omissão os tipos não têm um construtor de tipos definido.
 - ▶ Só pode ser definido, no máximo um construtor de tipo por cada tipo.
 - ▶ Têm de ser privados
 - ▶ Em C# a acessibilidade `private` é colocada automaticamente
 - ▶ O método não recebe parâmetros;

Iniciação de tipos - Exemplos

```
internal sealed class SomeRefType{  
    static SomeRefType(){  
        //. . .  
    }  
}
```

```
internal struct SomeValType{  
    static SomeValType(){  
        //. . .  
    }  
}
```

- Os campos com expressões de iniciação na sua definição, são os primeiros a ser iniciados pelo construtor

```
internal sealed class SomeType{  
    private static Int32 s = 5;  
    static SomeRefType(){  
        s = 10;  
    }  
}
```



Políticas de iniciação de tipo

- ▶ O CLR garante a chamada ao construtor de tipo antes de qualquer acesso a um campo de tipo;
- ▶ Políticas de iniciação de tipos:
 - ▶ Imediatamente antes do primeiro **acesso a qualquer membro** – usada em C# quando **há construtores de tipo**;
 - ▶ Em qualquer altura desde que antes do primeiro **acesso a um campo de tipo** (atributo de *Metadata* `beforefieldinit` – política em C# quando **não for definido explicitamente um constructor de tipo**).

Construção de objectos

- ▶ Quando se cria uma instância de um *reference type*:
 - ▶ É **reservado, no *managed heap***, um bloco de memória com o número de *bytes* necessários para armazenar o objecto do tipo especificado.
 - ▶ **Inicia os membros *overhead* do objecto**. Cada instância tem associados dois membros adicionais usados pelo CLR na gestão do objecto.
 - ▶ O primeiro membro é um apontador para a tabela de métodos do tipo;
 - ▶ O segundo é o *SyncBlockIndex* (usado na sincronização).
 - ▶ É **chamado o construtor** de instância do tipo.
 - ▶ Por fim, devolve uma **referência para o objecto criado**.



Construção de objectos

- ▶ O CLR requer que todos os objectos sejam criados usando o operador *new*
 - ▶ emite a instrução IL *newobj*.
 - ▶ EX: `Employee e = new Employee("ConstructorParam1");`
- ▶ Os construtores de instância nunca são herdados
 - ▶ Não lhes pode ser aplicado os modificadores `virtual`, `new`, `override`, `sealed` e `abstract`
 - ▶ Em C#, se uma classe não definir explicitamente um construtor, o compilador gera um por omissão.

```
public class SomeType{  
    // ... Sem construtor de instâncias  
}
```



```
public class SomeType{  
    public SomeType() : base () {  
    }  
    // ...  
}
```



Construção de objectos – keyword this

```
private sealed class SomeType{  
    private Int32 m_x;  
    private String m_s;  
  
    public SomeType( ){  
        m_x = 5;  
        m_s= "Ola";  
    }  
    public SomeType( Int32 x) : this ( ){  
        m_x = 10;  
    }  
}
```

- **Nota:** Embora a maioria das linguagens compilem os construtores de forma a que seja chamado o construtor do tipo base, o CLR não obriga à existência desta chamada.



Iniciação de instâncias

▶ Construtor de instância:

- ▶ Tem nome especial → `.ctor`
- ▶ Podem existir várias sobrecargas.

▶ Comportamento do construtor:

1. Inicia os campos que têm expressões de iniciação na sua definição;
2. Chama o construtor da classe base;
3. Executa o seu código.

Sobrecarga de métodos

- ▶ Podem ser sobrecarregados se diferirem em:
 - ▶ Número de parâmetros;
 - ▶ Tipo dos parâmetros;
 - ▶ Tipo de retorno; (não em C#)
 - ▶ Passagem de parâmetro por valor ou referência.
- ▶ As regras CLS permitem sobrecarga se os métodos diferirem apenas em número ou tipo dos parâmetros.

O tipo valor Ponto em C# (com redefinição de Equals)

```
public struct Ponto {  
    public int x, y;  
    public Ponto(int x, int y) { this.x=x; this.y=y; }  
  
    public override bool Equals(object obj) {  
        if (obj == null) return false;  
        if (!(obj is Ponto)) return false;  
        return Equals( (Ponto) obj);  
    }  
  
    public bool Equals(Ponto p) { return x== p.x && y == p.y; }  
    public override int GetHashCode() { return x^y; }  
    public override string ToString() {  
        return String.Format("({0},{1})", x, y);  
    }  
}
```

Desambiguar colisões de nomes

- ▶ Campos de tipo:

- ▶ Através do nome do tipo pretendido.

- ▶ Campos de instância:

- ▶ Através do uso das palavras **this** e **base**;

- ▶ Métodos:

- ▶ 2 políticas:

- ▶ *Hide-by-name* – esconde todos as sobrecargas de métodos com o mesmo nome; (C++)
 - ▶ *Hide-by-signature* – esconde o método com igual assinatura. (C#)

- ▶ Na CLS não existe colisão em nomes que variam apenas na capitalização dos caracteres.

Operador new - Exemplo

```
using System;

class A {
    public virtual void F() { Console.WriteLine("A.F"); } }

class B:A {
    public override void F() { Console.WriteLine("B.F"); } }

class C: B {
    new public virtual void F() { Console.WriteLine("C.F"); } }

class D: C {
    public override void F() { Console.WriteLine("D.F"); } }

class Test {
    static void Main() {
        D d = new D();
        A a = d; B b = d; C c = d;
        a.F(); b.F(); c.F(); d.F();
    }
}
```

As classes C e D
contêm **dois**
métodos com a
mesma assinatura!
D **redefine o**
método
introduzido em C

B.F
B.F
D.F
D.F

Operador new - Exemplo

```
using System;
```

```
class A {  
    public virtual void F() {} }
```

```
class B:A {  
    new private void F() {} // Esconde A.F em B  
}
```

```
class C: B {  
    public override void F() {} // Ok, redefine A.F  
}
```


Atributos aplicáveis a campos

IL Term	C# Term	Visual Basic Term	Description
static	static	Shared	Campo de tipo
initonly	readonly	ReadOnly	Só pode ser iniciado num construtor

- ◆ O CLR permite que um campo seja marcado como `static`, `initonly` ou `static e initonly`.
- ◆ O C# suporta a combinação dos dois.

Constantes – C#

- É um símbolo ao qual é atribuído a um valor que nunca se altera.
- O valor a atribuir têm que ser determinado em tempo de compilação, logo, **apenas** podem ser **definidas constantes de tipos primitivos da linguagem**.
- O **compilador** guarda o valor da constante na `Metadata` do módulo (como um campo **estático literal**)
 - não é alocada memória em tempo de execução
- Não é possível obter o endereço duma constante nem passá-la por referência.
- Por levantar problemas de versões, só devem ser usadas quando existe certeza que o seu valor é imutável (p.ex. `Pi`, `MaxInt16`, `MaxInt32`, etc.).
 - Exemplo em C#
 - **`const Int32 SomeConstant = 1;`**

Campos readonly – C#

- **Campos readonly**
 - Só podem ser afectados durante a construção da instância do tipo onde estão definidos
 - Um campo static readonly é definido em *run time*.
 - Uma constante é definida em *compile time*.
 - Em C#: **readonly Int32 SomeReadOnlyField = 2;**
 - O campo é marcado com o atributo *InitOnly*

Propriedades

- ▶ As propriedades permitem ao código fonte invocar um método utilizando uma sintaxe simplificada.
- ▶ Existem dois tipos de propriedades:
 - ▶ Propriedades sem parâmetros
 - ▶ Propriedades com parâmetros
 - ▶ C# designa as propriedades com parâmetros por *indexers*.

Propriedades sem parâmetros

```
public sealed class Employee{
    private String m_Name;
    private Int32 m_Age;
    public String Name {
        get { return( m_Name); }
        set { m_Name = value; }
    }
    public Int32 Age {
        get { return( m_Age); }
        set { if ( value < 0 )
                throw new
                ArgumentOutOfRangeException("value",
                    value.ToString(),
                    "O valor tem de ser maior ou igual a 0");
            m_Age = value;}
    }
}
```

```
Employee e = new Employee();
String empName = e.Name;
e.Age = 41;
Int32 i = e. Age;
e.Age = -5;
```

Propriedades sem parâmetros

- ▶ Cada propriedade tem um **nome e um tipo**;
- ▶ **Não** pode existir **sobrecarga** de propriedades;
- ▶ Acedidas através de um nome ou um acesso de membro
- ▶ Pode ser um membro **estático ou de instância**
- ▶ O get accessor de uma propriedade não tem parâmetros
- ▶ O set accessor de uma propriedade contém implicitamente o **parâmetro value**.

Propriedades sem parâmetros - Exemplo 2

```
public sealed class Employee{
    private String m_Name;
    private Int32 m_Age;
    private static Int32 nR_Employees;
    public String Name {
        get { return(m_Name); }
        set { m_Name = value; }
    }
    public Int32 Age {
        get { return( m_Age); }
        set { if ( value < 0 ) throw new ArgumentOutOfRangeException("value",
                                                                    value.ToString(),
                                                                    "O valor tem de ser maior ou igual a 0");
            m_Age = value;}
    }
    public static Int32 NrEmployees{
        get { return(nR_Employees); }
    }
}
```



Propriedades estáticas e de instância

```
using System;
using CTSTester.Properties;

namespace CTSTester {
    namespace Properties {
        public class TypeWithProps {
            private static int aTypeField;

            public string AnInstanceProperty {
                get { return "instance property"; }
            }

            public static int ATypeProperty {
                get { return aTypeField; }
                set { aTypeField = value; }
            }
        }
    }

    class TestProperties {
        public static void Main() {
            TypeWithProps mt = new TypeWithProps();
            System.Console.WriteLine(mt.AnInstanceProperty);
            System.Console.WriteLine(TypeWithProps.ATypeProperty);
            TypeWithProps.ATypeProperty = 30;
            System.Console.WriteLine(TypeWithProps.ATypeProperty);
        }
    }
}
```


Propriedades estáticas e de instância (IL de Main)

```
.method public hidebysig static void Main() cil managed {
    .entrypoint
    // Code size          45 (0x2d)
    .maxstack 2
    .locals init ([0] class CTSTester.Properties.TypeWithProps mt)
        newobj          instance void CTSTester.Properties.TypeWithProps::.ctor()
        stloc.0
        ldloc.0
        callvirt        instance string
CTSTester.Properties.TypeWithProps::get_AnInstanceProperty()
        call            void [mscorlib]System.Console::WriteLine(string)
        call            int32
CTSTester.Properties.TypeWithProps::get_ATypeProperty()
        call            void [mscorlib]System.Console::WriteLine(int32)
        ldc.i4.s        30
        call            void
CTSTester.Properties.TypeWithProps::set_ATypeProperty(int32)
        call            int32
CTSTester.Properties.TypeWithProps::get_ATypeProperty()
        call            void [mscorlib]System.Console::WriteLine(int32)
        ret
} // end of method TestProperties::Main
```



Propriedades com parâmetros

- ▶ Identificado pela sua **assinatura**.
- ▶ Acedido através de um acesso de um elemento.
- ▶ Tem de ser um **membro de instância**.
- ▶ O **get** accessor de um *indexer* tem o **mesma lista de parâmetros formais** que um *indexer*.
- ▶ O **set** accessor de um *indexer* tem a **mesma lista de parâmetros formais de** um *indexer*, assim como o parâmetro **value**.

Indexers (criação)

```
class IndexerClass {  
    private int [] myArray = new int[100];  
    public int this [int index] {  
        get {  
            if (index < 0 || index >= 100) return 0;  
            else return myArray[index];  
        }  
        set {  
            if (!(index < 0 || index >= 100))  
                myArray[index] = value; }  
        }  
    }  
}
```

Indexers (Utilização)

```
public class MainClass {  
    public static void Main() {  
        IndexerClass b = new IndexerClass();  
        b[3] = 256;  
        b[5] = 1024;  
        for (int i=0; i<=10; i++) {  
            Console.WriteLine("Element #{0} = {1}", i, b[i]);  
        }  
    }  
}
```

Classes parciais – C#

- ▶ **Objectivo:**
 - ▶ Separar o código gerado automaticamente do código escrito pelo programador
- ▶ Uma classe pode ser dividida em partes, em que cada parte corresponde a uma implementação *parcial* da classe.
- ▶ Todas as partes da classe devem estar disponíveis no momento da compilação
 - ▶ gera uma única classe em representação intermédia
 - ▶ classe reside num único *assembly*

Classes parciais – C#

- ▶ Aspectos acumulativos de uma classe:
 - ▶ Campos
 - ▶ Métodos
 - ▶ Propriedades
 - ▶ Indexadores
 - ▶ Interfaces implementadas
- ▶ Aspectos não acumulativos:
 - ▶ Classe base
 - ▶ Tipo-valor ou tipo-referência
 - ▶ Visibilidade
- ▶ As diversas partes de uma mesma classe devem concordar nos aspectos não acumulativos.

Enumerados

- ▶ Os **enumerados** permitem definir especializações de tipos integrais que não adicionam campos, mas simplesmente restringem o espaço de valores de um tipo integral específico.
- ▶ O tipo enumerado é um **tipo valor** em que o tipo base é **System.Enum**.
- ▶ Os enumerados têm um **segundo tipo subjacente** que irá ser usado para a representação dos dados da enumeração
 - ▶ Se não for especificado nenhum, é assumido pelo **compilador C#** o **tipo System.Int32**.
 - ▶ O Tipo **System.Char** não pode ser utilizado.
- ▶ Os enumerados pode ser representados em formas *boxed* e *unboxed*.
- ▶ **Não** podem definir métodos, propriedades ou eventos.

Enumerados

```
enum EstacoesDoAno {  
    Primavera, Verao, Outono, Inverno  
};
```

```
.class private auto ansi sealed Geometry.EstacoesDoAno  
    extends [mscorlib]System.Enum {  
    .field public specialname rtspecialname int32 value__  
  
    .field public static literal valuetype  
        EstacoesDoAno Primavera = int32(0x00000000)  
    .field public static literal valuetype  
        EstacoesDoAno Verao = int32(0x00000001)  
    .field public static literal valuetype  
        EstacoesDoAno Outono = int32(0x00000002)  
    .field public static literal valuetype  
        EstacoesDoAno Inverno = int32(0x00000003)  
} // end of class Geometry.EstacoesDoAno
```



Exemplo - Continuação

```
using System;

internal enum Color{
    White,
    Red,
    Green,
    Blue,
    Orange }

public class Program{
    public static void Main(){
        Console.WriteLine(Color.Format(typeof(Color),3,"G"));
        Color c = (Color) Enum.Parse(typeof(Color), "White");
        Console.WriteLine(c);
        Console.WriteLine("{0:D} \t {0:G}",c);
        c = (Color) Enum.Parse(typeof(Color), "white",true);
        Console.WriteLine(c.ToString("G"));
        Console.WriteLine(Enum.IsDefined(typeof(Color),"red"));
    }
}
```

Blue
White
0 White
White
False



O tipo Enum (excerto)

<code>int CompareTo(object target)</code>	Compara esta instância com o objecto especificado e retorna uma indicação dos seus valores relativos.
<code>static string Format(Type enumType, object value, string format)</code>	Converte o valor especificado de um tipo enumerado especificado para a sua representação equivalente sob a forma de string, de acordo com o formato especificado.
<code>static string GetName(Type enumType, object value)</code>	Retorna o nome da constante na enumeração especificada que tem aquele valor especificado.
<code>static string[] GetNames(Type enumType)</code>	Retorna um array com os nomes das constantes na enumeração especificada.
<code>static Type GetUnderlyingType(Type enumType)</code>	Retorna o tipo subjacente da enumeração especificada.
<code>static Array GetValues(Type enumType)</code>	Retorna um array com os valores das constantes numa enumeração especificada.
<code>static bool IsDefined(Type enumType, object value)</code>	Retorna true se e só se uma constante com um valor especificado existe numa enumeração especificada.
<code>static object Parse(Type enumType, string value)</code>	Converte a representação sob a forma de string do nome ou valor numérico de uma ou mais constantes enumeradas para um objecto enumerado equivalente.



Interfaces

- ▶ Keyword `interface` (para definição de “*Interface Types*”)
 - ▶ Para especificação de contratos, isto é, conjunto de operações suportadas
- ▶ As interfaces suportam herança múltipla de outras interfaces.
- ▶ Não podem conter campos de instância nem métodos de instância com implementação.
- ▶ Todos os métodos de instância têm, implicitamente, os atributos `public` e `virtual`.
- ▶ Por convenção, o nome das interfaces começa pelo carácter ‘I’
 - ▶ Exemplos: `ICloneable`, `IEnumerable`

Exemplo – interface A

```
public interface A{  
    void GetA();  
}
```



```
.class interface public abstract auto ansi A  
{  
    .method public hidebysig newslot abstract virtual  
        instance void  GetA() cil managed  
    {  
    } // end of method A::GetA  
  
} // end of class A
```

Interface IComparable

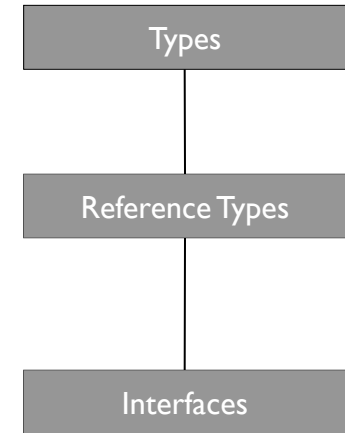
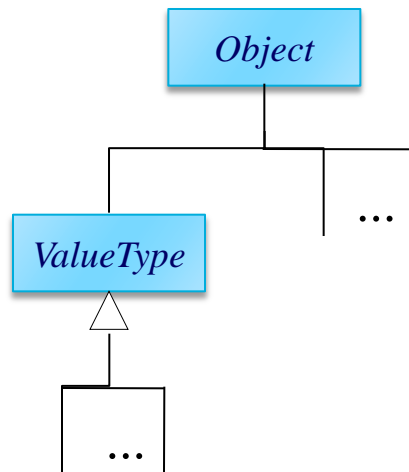
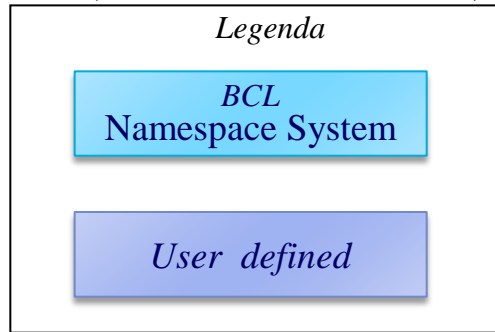
```
public interface IComparable<T>{  
    Int32 CompareTo(T other);  
}
```

```
public class Point : IComparable<Point>{  
    private Int32 x,y;  
    public Point( Int32 x, Int32 y){  
        this.x=x;  
        this.y=y;  
    }  
    public Int32 CompareTo(Point other){  
        return Math.Sign(Math.Sqrt(x*x+y*y) -  
            Math.Sqrt(other.x*other.x + other.y*other.y)  
        );  
    }  
}
```

Interfaces

- ▶ Em C# a implementação de uma interface resulta, por omissão, em métodos *sealed*.
- ▶ O que não acontece se o método na classe for declarado como *virtual*.
- ▶ Exemplo:
 - ▶ Em C#
 - `public Int32 CompareTo(Point other){...}`
 - ▶ Em IL:
 - `.method public hidebysig newslot virtual final
instance int32 CompareTo(class Point
other) cil managed
{`

Linguagem C# - Excerto do modelo de tipos (interfaces)



Interfaces – Exemplo 2

```
using System;
using System.Collections;

public class Program{

public static void Main(){
    String s="AVE";
    ICloneable cloneable = s;
    IComparable comparable = s;
    IEnumerable enumerable = (IEnumerable) comparable;
}
}
```

Porque o tipo do objecto
implementa ambas as interfaces



Interfaces (pré-genéricos) de enumeração - IEnumerable, IEnumerator e C# *foreach*

```
public interface System.Collections.IEnumerable {  
    IEnumerator GetEnumerator();  
}  
public interface System.Collections.IEnumerator {  
    Boolean MoveNext();  
    void Reset();  
    Object Current { get; }  
}
```

```
ArrayList vals = new ArrayList(new Int32[]{ 1, 4, 5 });  
IEnumerator itr = vals.GetEnumerator();  
...  
while(itr.MoveNext()) Console.WriteLine((Int32)itr.Current);  
...
```

```
ArrayList vals = new ArrayList(new Int32[]{ 1, 4, 5 });  
foreach(Int32 v in vals) Console.WriteLine(v);
```



ICloneable

```
public interface System.ICloneable {  
    Object Clone();  
}
```

- ◆ Implementada por tipos que permitam “clonagem” de instâncias
- ◆ Políticas de “clonagem”:
 - Cópia superficial (*shallow copy*)
 - Método `Object.MemberwiseClone()` de `System.Object`
 - Cópia total (*deep copy*)



Interfaces e *Value Types*

- ◆ Os *Value Types* derivam obrigatoriamente e directamente de `System.ValueType` ou de `System.Enum` mas podem implementar 0 ou mais interfaces (via *boxing*)
- ◆ A conversão entre um *Value Type* e as interfaces por ele implementadas está sujeita às regras de conversão entre *Value Types* e *Reference Types* (*boxing* e *unboxing*)

```
public struct Point : System.IComparable {  
    public Int32 CompareTo(Object o) { return ... }  
}  
  
...  
Point p1 = new Point(1,2), p2 = new Point(2,1);  
p1.CompareTo(p2); // boxing em p2  
((System.IComparable)p1).CompareTo(p2); // boxing em p1 e p2  
...
```



Implementação Explícita de interfaces

- Uma classe que implementa uma interface pode explicitamente implementar um membro dessa interface.
 - Quando um membro é explicitamente implementado, não pode ser acessado através uma instância da classe.

Exemplo:

```
interface IDimensions { float Length(); float Width(); }
```

```
class Box : IDimensions {  
    float lengthInches; float widthInches;  
    public Box(float length, float width) {  
        lengthInches = length; widthInches = width;  
    }  
    float IDimensions.Length() { return lengthInches; }  
    float IDimensions.Width() { return widthInches; }  
    public static void Main() {  
        Box myBox = new Box(30.0f, 20.0f);  
        IDimensions myDimensions = (IDimensions) myBox;  
        System.Console.WriteLine("Length: {0}", myDimensions.Length());  
        System.Console.WriteLine("Width: {0}", myDimensions.Width());  
    }  
}
```



Implementação Explícita de interfaces(2)

```
interface IEnglishDimensions { float Length(); float Width(); }
```

```
interface IMetricDimensions { float Length(); float Width(); }
```

```
class Box : IEnglishDimensions, IMetricDimensions {  
    float lengthInches; float widthInches;  
    public Box(float length, float width) {  
        lengthInches = length; widthInches = width;  
    }  
    float IEnglishDimensions.Length() { return lengthInches; }  
    float IEnglishDimensions.Width() { return widthInches; }  
    float IMetricDimensions.Length() { return lengthInches * 2.54f; }  
    float IMetricDimensions.Width() { return widthInches * 2.54f; }  
    public static void Main() {  
        Box myBox = new Box(30.0f, 20.0f);  
        IEnglishDimensions eDimensions = (IEnglishDimensions) myBox;  
        IMetricDimensions mDimensions = (IMetricDimensions) myBox;  
        System.Console.WriteLine("Length(in): {0}", eDimensions.Length());  
        System.Console.WriteLine("Length(cm): {0}", mDimensions.Length());  
    }  
}
```

A implementação explícita de um membro de interface pode ser útil, por exemplo, em cenários de implementação de duas interfaces que partilham métodos com a mesma assinatura

Implementação explícita de Interfaces

- ▶ Os tipos que implementam as interfaces podem dar uma implementação explícita de alguns métodos da interface.
 - ▶ A implementação explícita é privada

```
public interface IA {  
    void Method1();  
    void Method2(Int32 val);  
}
```

```
public class Impl : IA {  
    public void Method1(){  
        ..  
    }  
    void IA.Method2(Int32 val){  
        ..  
    }  
}
```

Implementação explícita de interfaces

- Um tipo exacto (classe) pode optar por esconder a implementação de um método da sua “interface” pública

```
AClass a = new AClass();  
a.Draw();    // AClass.Draw  
IWindow iw = (IWindow) a;  
iw.Draw(); // IWindow.Draw
```

```
public interface IWindow { void Draw(); }  
public interface IArtist { void Draw(); }  
  
public class AClass : IWindow , IArtist {  
    // apenas visível com uma referência para ICowboy  
    void IWindow.Draw() { }  
    // apenas visível com uma referência para IArtist  
    void IArtist.Draw() { }  
    // visível com uma referência para aClass  
    public void Draw() { }  
}
```

Implementação explícita de Interfaces (II)

A implementação explícita de interfaces permite evitar operações de *box* e *unbox* na invocação de métodos de implementação de interfaces em tipos valor, usando o idioma mostrado a seguir:

```
struct Val : ICloneable {  
    public int v;  
  
    object ICloneable.Clone() {  
        return MemberwiseClone();  
    }  
  
    public Val Clone() {  
        return new Val(v);  
    }  
}
```

Usado quando:

```
ICloneable ic = new Val(5);  
Val v1 = (Val) ic.Clone();
```

Permite:

```
Val v = new Val(5);  
Val v1 = v.Clone();  
Sem operações de box e unbox
```


Linguagem C# - Modelo de tipos revisitado

