
CTS via C#

Parte 2

Agenda

- Membros de tipos
- Operadores
- Invocação de métodos
- *Explicit Interface Method Invocation* (EIMI)

Agenda

- **Membros de tipos**
- Operadores
- Invocação de métodos
- *Explicit Interface Method Invocation (EIMI)*

Membros de tipos

- Um tipo pode definir os seguintes membros:
 - Constantes
 - Campos
 - Construtores de instância
 - Construtores de tipo (construtores estáticos)
 - Métodos
 - Sobrecarga de operadores (*operator overloading*)
 - Operadores de conversão
 - Propriedades
 - Eventos
 - Tipos aninhados

Membros de tipos

- Constantes
 - É um símbolo que identifica um valor que nunca se altera
 - Estão sempre associados ao tipo (não às instâncias) e como tal são compilados como membros estáticos
- Construtores de tipo
 - Construtores usados para iniciar os campos estáticos
- Sobrecarga de operadores
 - São métodos que definem como os objectos devem ser manipulados quando lhes são aplicados certos operadores especiais (ex: +, -)
 - Não é suportado no CLS
- Operadores de conversão
 - São métodos que definem como um objecto dum tipo deve ser convertido para outro tipo
 - Não é suportado no CLS

Membros de tipos

- Propriedades

- Define um mecanismo que permite, de forma simplificada, usando a sintaxe de manipulação de campos, aceder para leitura/escrita ao estado de um tipo (propriedades estáticas) ou de uma instância (propriedades de instância)

- Eventos

- Um **evento estático** é mecanismo que permite a um **tipo enviar uma notificação** a outro tipo ou objectos (ouvintes)
- Um **evento de instância** (não estático) é um mecanismo que permite a um **objecto enviar uma notificação** a outro tipo ou objectos
- Os eventos são normalmente gerados em resposta a uma alteração do estado ocorrido no tipo ou objecto que disponibiliza o evento.

Membros de tipos

- Eventos

- O registo num dado evento (realizado por ouvintes - *listeners*) é feito através de dois métodos que permitem tipos ou objectos registarem ou revogarem o registo
 - `add_NomeEvento`
 - `remove_NomeEvento`
- Em adição, os eventos usam tipicamente um campo *delegate* para manter o conjunto dos *listeners* registados.

Visibilidade de tipos e acessibilidade de membros

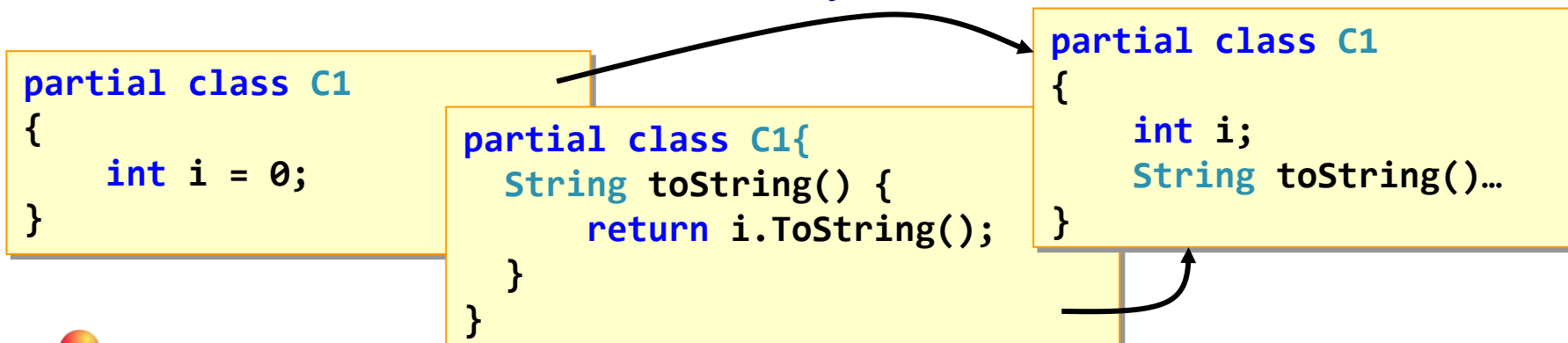
- Visibilidade de tipos
 - `internal` – visibilidade por omissão, o tipo só é visível dentro do assembly onde está definido
 - `public`
 - Assemblies “friend”
 - Permite especificar um assembly “amigo” que pode aceder a tipos internos ao assembly fonte
- Acessibilidade de membros
 - `private`
 - `internal` - visível dentro do assembly onde o tipo está definido
 - `protected` - visível em classes derivadas (definidas no mesmo ou noutros assemblies)
 - `protected internal` - visível em classes derivadas ou no assembly
 - `public`

Constantes e campos *readonly*

- Constantes
 - O valor armazenado tem que ser determinado em tempo de compilação
 - Não é possível obter o endereço duma constante nem passá-la como parâmetro por referência
 - Armazenada na metadata de um módulo, na forma de um campo estático
 - Ex: `public const Int32 MaxEntriesInList = 50;`
- A referência a uma constante definida num tipo de outro módulo faz com que o seu valor seja embebido directamente na metadata do módulo que faz uso dessa constante
 - Ex: `ldc.i4.s 50` // em vez de aceder ao campo `MaxEntriesInList`
- Campos *readonly*
 - Apenas pode ser iniciado num construtor

Classes parciais

- O objectivo das classes parciais é a separação do 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
 - é gerada uma única classe em representação intermédia
 - a classe reside num único *assembly*



Classes parciais (cont.)

- 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.

Membros de tipos

- Classes estáticas
 - No .NET 2.0 existem também classes estáticas
 - Não podem ser instanciadas
 - Só podem ter membros estáticos e são automaticamente *sealed* (não admitem derivação)
 - Ex: classe `Math`, que contém campos estáticos (`PI`, `E`, ...) e métodos utilitários (`Cos`, `Sin`, ...)
- Tipos aninhados (*Nested types*)
 - Em C# uma classe aninhada (não estática) não tem acesso aos membros de instância da classe mãe (*enclosing class*)
 - É possível criar instâncias de classes aninhadas não estáticas

```
class A {  
    class B { }  
}
```

Construtores de instância

- Método de instância especial com o nome `.ctor`
- Recebe como 1.º argumento a referência para o objecto a ser iniciado (`this`)
- Cuidados: não devem ser iniciados campos de instância na sua declaração

```
class A {  
    int i = 5;  
    double d = 3.7;  
    string s = "Olá";  
    public A(int i, double d, string s) {  
        // AQUI e em TODOS os construtores é colocado  
        // o código de iniciação (acima) dos campos  
        this.i = i;  
        this.d = d;  
        this.s = s;  
    }  
}
```

Solução: Colocar o código de iniciação no `.ctor` sem parâmetros

Construtores de tipo (estáticos)

- Não recebem parâmetros
- Não podem ser chamados explicitamente, sendo privados por omissão
 - Quem decide quando chamar o construtor estático é o VES
 - O VES invoca o construtor de tipo imediatamente antes de a primeira instância do tipo ser criada ou antes de qualquer método estático ser invocado.

```
public class A {  
    public static int i;  
    static A() { Console.WriteLine("Chamada"); }  
}
```

Main:

```
Console.WriteLine("started"); Console.ReadKey();  
// Só aqui é chamado o construtor estático  
Console.WriteLine(A.i);
```

Construtores de tipo (estáticos)

- Se tivermos:

```
public class A {  
    public static int i = 10;  
}
```

- O compilador gera automaticamente um construtor estático
- Esta forma dá mais flexibilidade ao VES dado que este gera a chamada no início do método que manipula o membro estático
- A forma explícita (apresentada anteriormente) pode ser menos eficiente pois obriga o VES a realizar a chamada imediatamente antes do membro ser acedido (problemático se estiver dentro dum ciclo **for**)

Passagem de parâmetros por referência

```
public sealed class Program {  
    public static void Swap(String s1, String s2) {  
        String aux = s1;  
        s1 = s2;  
        s2 = aux;  
    }  
  
    public static void Main() {  
        String a1 = "str 1";  
        String a2 = "str 2";  
        Swap(a1, a2);  
        Console.WriteLine("s1 = " + a1 + "; s2 = " + a2);  
    }  
}
```

- Não troca as *strings* porque as referências passadas à função são passadas por cópia.

Passagem de parâmetros por referência

```
public sealed class Program {  
    public static void Swap_ok(ref String s1, ref String s2) {  
        String aux = s1;  
        s1 = s2;  
        s2 = aux;  
    }  
    public static void Main() {  
        String a1 = "str 1";  
        String a2 = "str 2";  
        //Swap(a1, a2);  
        Swap_ok(ref a1, ref a2);  
        Console.WriteLine("s1 = " + a1 + "; s2 = " + a2);  
    } }  
}
```

- Tem que se passar à função uma referência para a referência para string (note-se a adição do atributo ref na declaração e na chamada)

Passagem de parâmetros por referência

- Atributo **out**
 - Para passar um parâmetro que não deve ser consumido mas apenas afectado, é usada a palavra **out** antes do parâmetro

```
class Program {  
    static void GetVal(out int v) { ... }  
    static void AddVal(ref int v) { ... }  
  
    public static void Main() {  
        int n;  
        AddVal(ref n); // Erro, variável não iniciada  
        GetVal(out n); // OK  
        AddVal(ref n); // OK  
    }  
}
```

Lista de parâmetros variável

```
public static void WriteMany(params String[] v) {  
    // Mostrar entradas de v  
}  
public static void Main(String[] args) {  
    WriteMany("aaa");  
    WriteMany("aaa", "bbb", "ccc");  
    // Transforma em array de Strings  
}
```

- Definidos com o atributo `params`
- Gera MSIL para instanciar e preencher *array*
- É recomendado ter sobrecarga de 2 ou 3 métodos que recebam os argumentos separadamente:
 - `ShowArgs(arg1)`
 - `ShowArgs(arg1, arg2)`
- O tipo do array pode ser qualquer (ex: `double`, `object`)

Propriedades

```
public class Point {  
    public double x, y;  
    ...  
    public double Abs {  
        get { return Math.Sqrt(x*x + y*y); }  
        set {  
            double phase = Phase; // Phase: outra propriedade  
            x = value * Math.Cos(phase);  
            y = value * Math.Sin(phase);  
        }  
    }  
    public static void Main() {  
        Point p = new Point();  
        p.Phase = 30; // São chamados o set de Phase e Abs  
        p.Abs = 10;   // passando no value os valores 30 e 10  
        Console.WriteLine("p = (" + p.x + ";" + p.y + ")");  
        Console.WriteLine("p.Abs = " + p.Abs);  
        Console.WriteLine("p.Phase = " + p.Phase);  
    }  
}
```

Agenda

- Membros de tipos
- **Operadores**
- Invocação de métodos
- *Explicit Interface Method Invocation (EIMI)*

Overload de operadores

- O CLR desconhece a sobrecarga de operadores
- O compilador faz a tradução entre os operadores e a chamada a determinados métodos especiais

```
class Program {  
    public static void Main() {  
        Object o1 = new Object();  
        Object o2 = new Object();  
        bool res1 = o1 == o2;  
        // ceq -> false  
        A a1 = new A();  
        A a2 = new A();  
        bool res2 = a1 == a2;  
        // ceq ou operator==  
        Console.WriteLine("res1 = " + res1);  
        Console.WriteLine("res2 = " + res2);  
    }  
}
```

Overload de operadores

- Exemplo: Sobrecarga do `operator==`

demo

```
class A {  
    private int i;  
  
    public static bool operator==(A a, A other) {  
        return a.i == other.i;  
    }  
  
    public static bool operator!=(A a, A other) {  
        return !(a == other);  
    }  
}
```

- É obrigatório que pelo um dos parâmetros seja do tipo da classe que define o operador

Agenda

- Membros de tipos
- Operadores
- **Invocação de métodos**
- *Explicit Interface Method Invocation (EIMI)*

Invocação de métodos

- Para cada tipo carregado em memória pelo VES é iniciada uma estrutura com informação de tipo (RTTI).
- Sempre que o *jitter* compila um método para nativo, são carregados todos os tipos referidos.
- Despacho dinâmico: chamada a um método com comportamento polimórfico (método virtual).
- Despacho estático: usado na invocação de métodos não virtuais.

Invocação de métodos em tipos referência (1)

- Sejam as seguintes definições de classes:

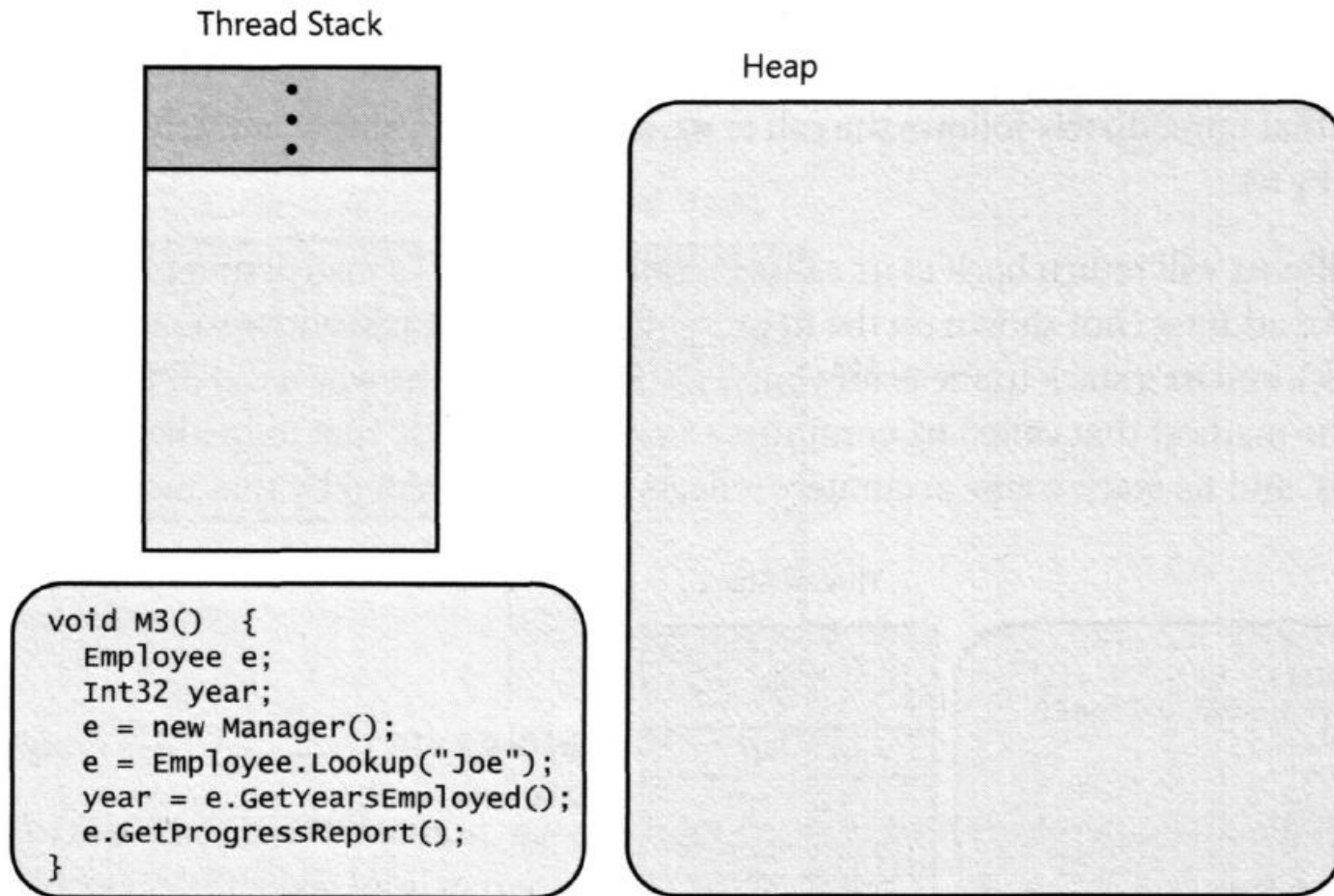
```
internal class Employee
{
    public Int32 GetYearsEmployed() { ... }
    public virtual String GetProgressReport() { ... }
    public static Employee Lookup(String name) { ... }
}

internal sealed class Manager : Employee
{
    public override String GetProgressReport() { ... }
}
```

Invocação de métodos em tipos referência (2)

- *Stack* antes da invocação de M3

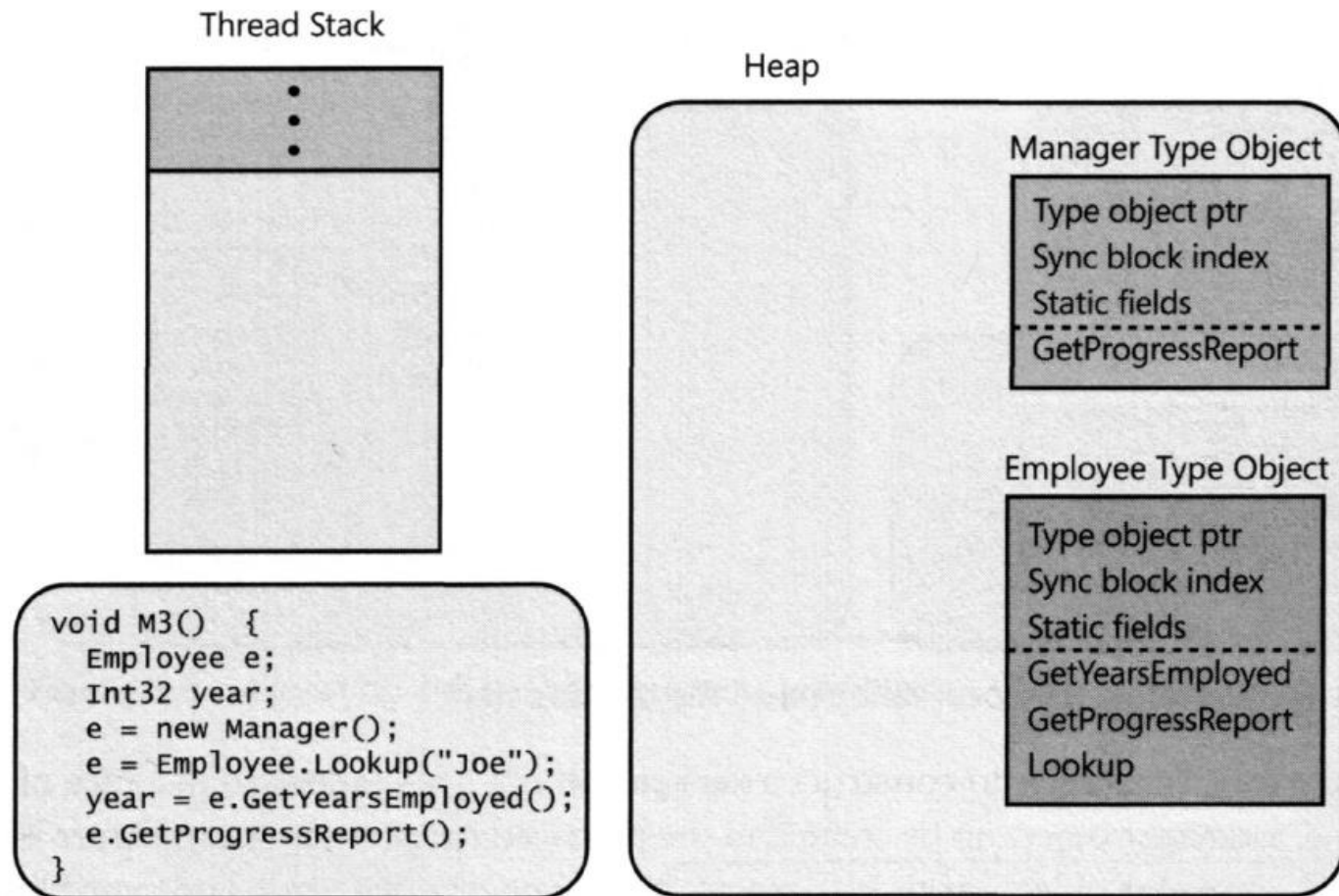
(adaptado de: “CLR via C#”, Jeffrey Richter)



Invocação de métodos em tipos referência (3)

- Tipos referenciados por M3: `Employee`, `Int32`, `Manager`, e `String`

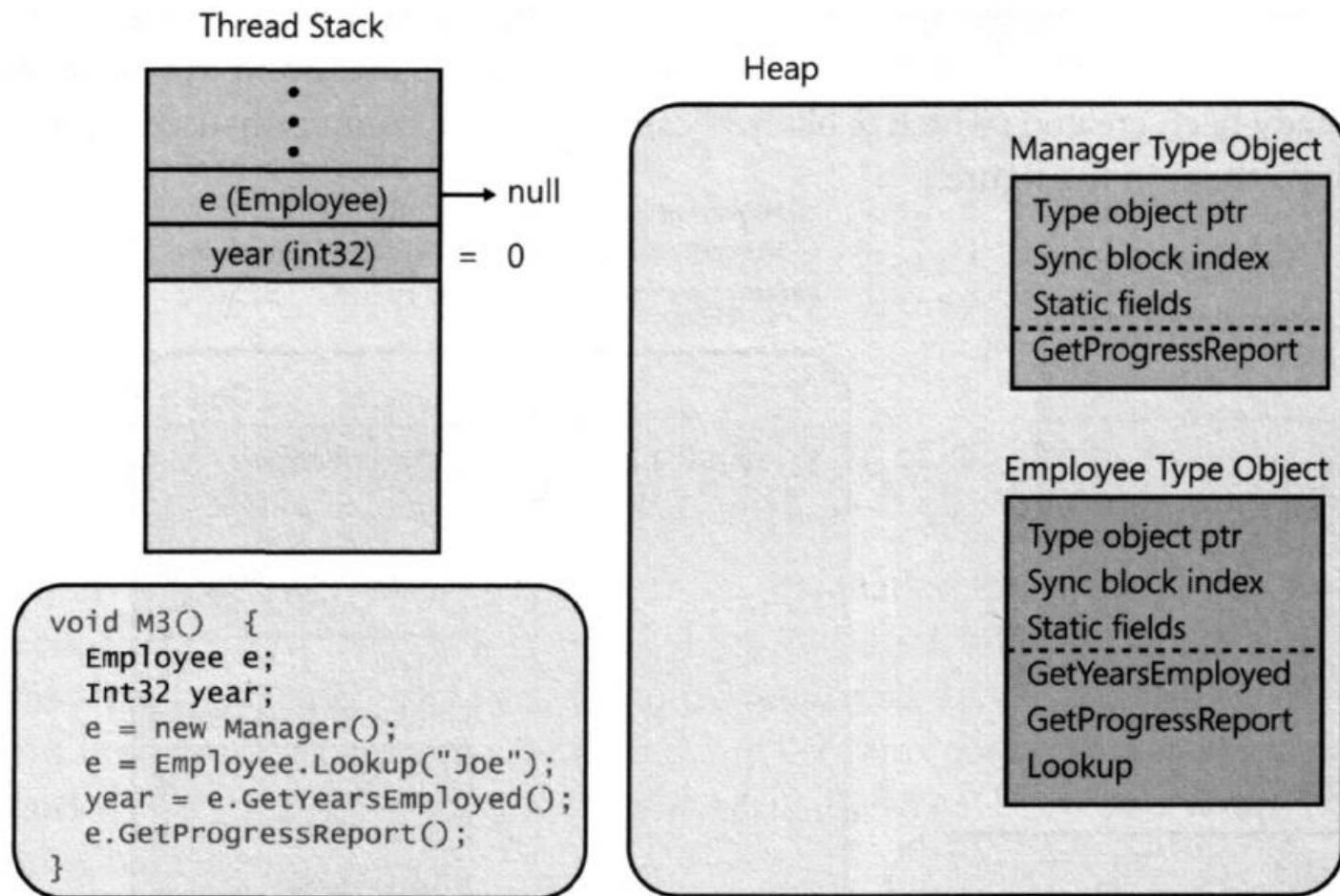
(adaptado de: “CLR via C#”, Jeffrey Richter)



Invocação de métodos em tipos referência (4)

- Aloja variáveis locais de **M3** no *stack*

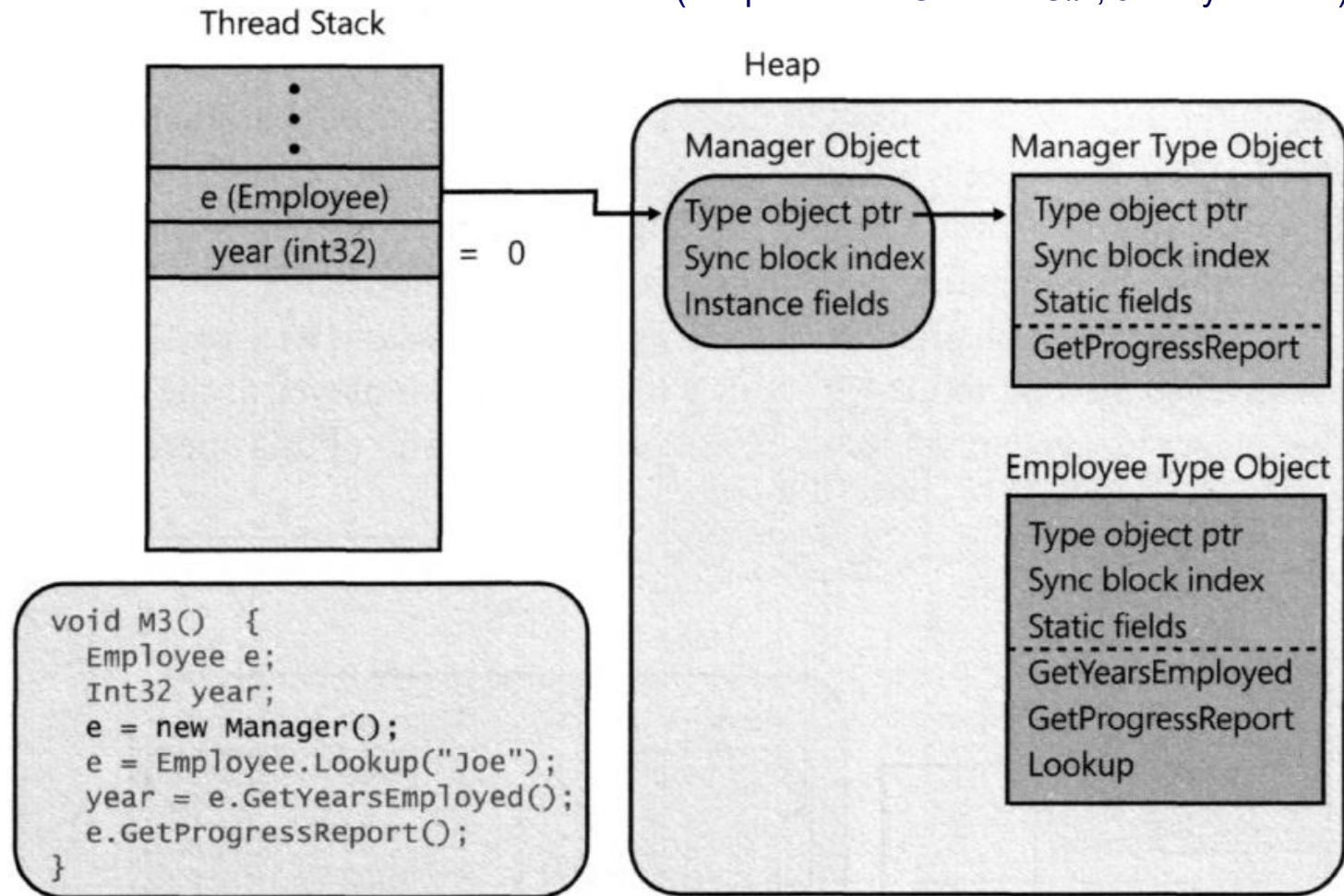
(adaptado de: "CLR via C#", Jeffrey Richter)



Invocação de métodos em tipos referência (5)

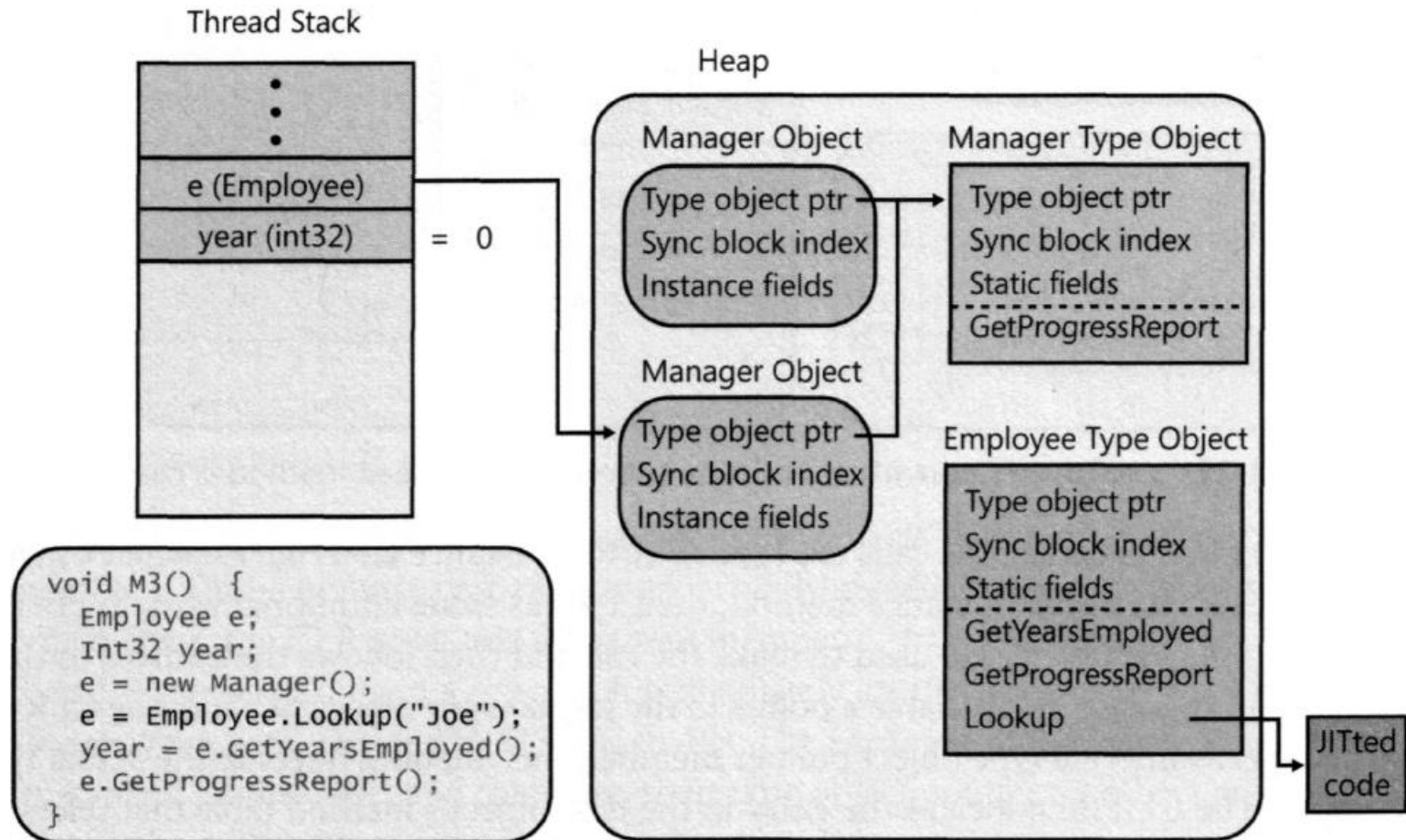
- Alojar e iniciar um objecto do tipo **Manager**

(adaptado de: "CLR via C#", Jeffrey Richter)



Invocação de métodos em tipos referência (6)

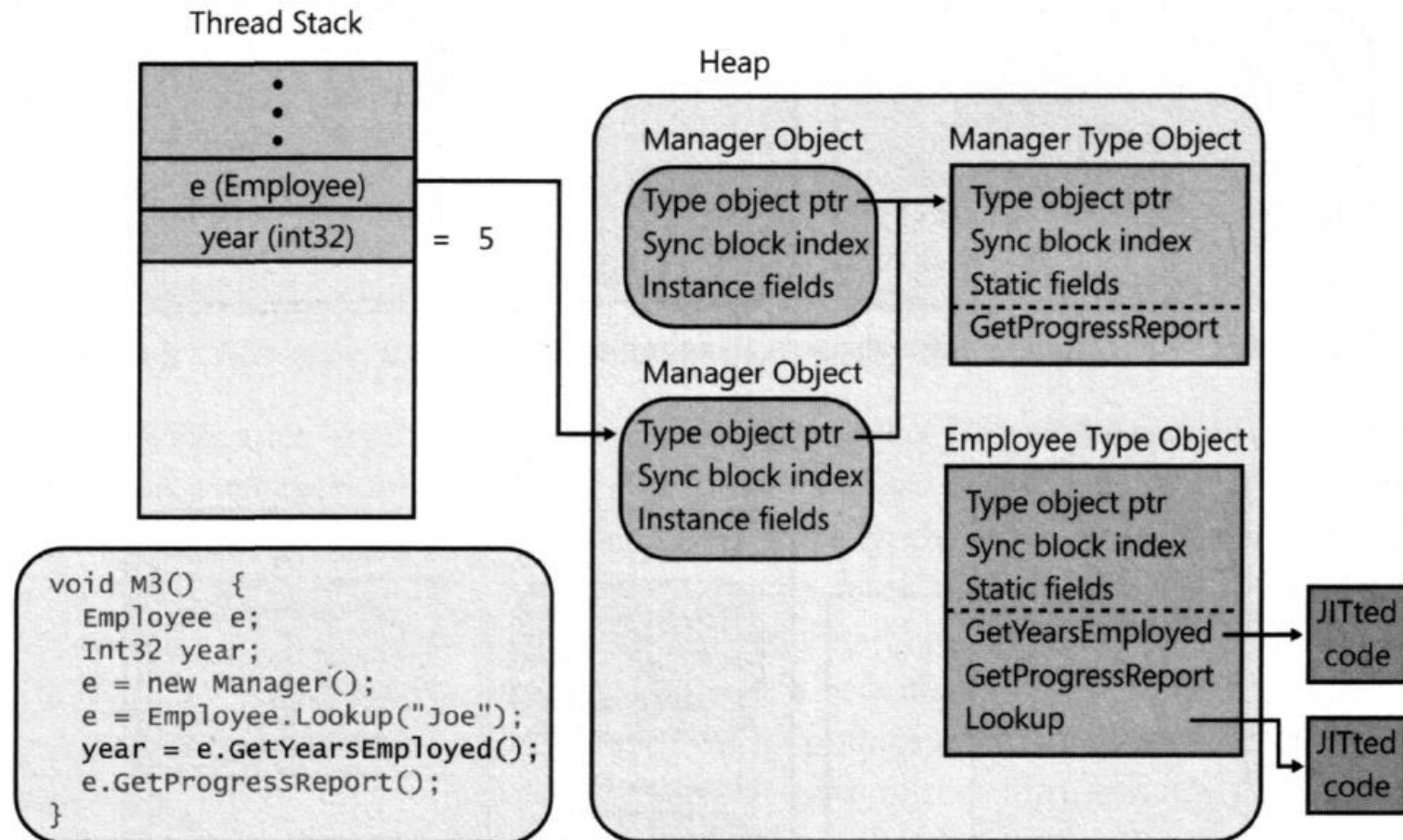
- O método estático `Lookup` de `Employee` aloja e inicia um objecto `Manager` que representa `Joe` (adaptado de: "CLR via C#", Jeffrey Richter)



Invocação de métodos em tipos referência (7)

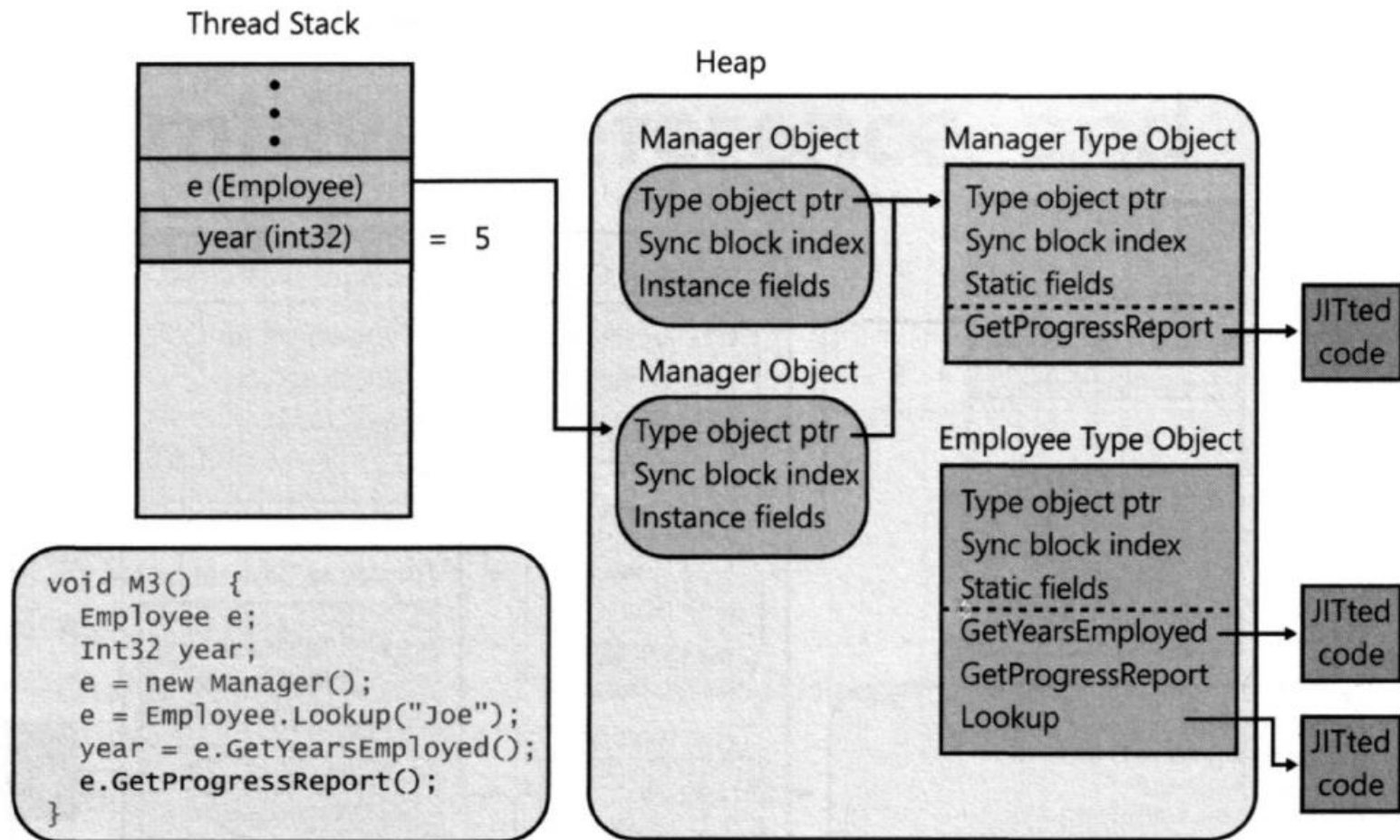
- O método de instância não virtual `GetYearsEmployed` de `Employee` é invocado, retornando 5

(adaptado de: "CLR via C#", Jeffrey Richter)

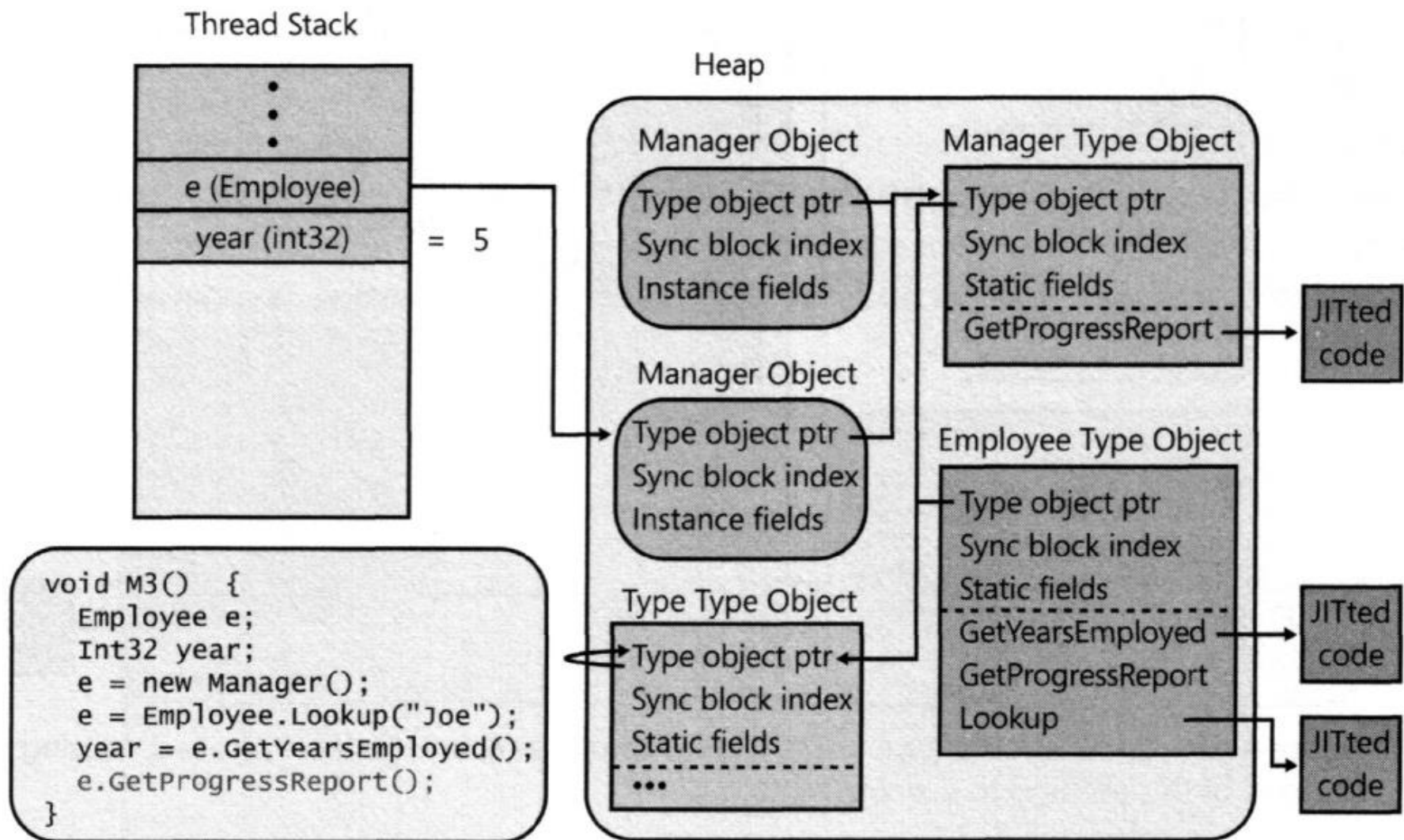


Invocação de métodos em tipos referência (8)

- O método virtual `GenProgressReport` de `Employee` é invocado, fazendo com que seja executado a versão redefinida em `Manager`



Invocação de métodos em tipos referência (9)

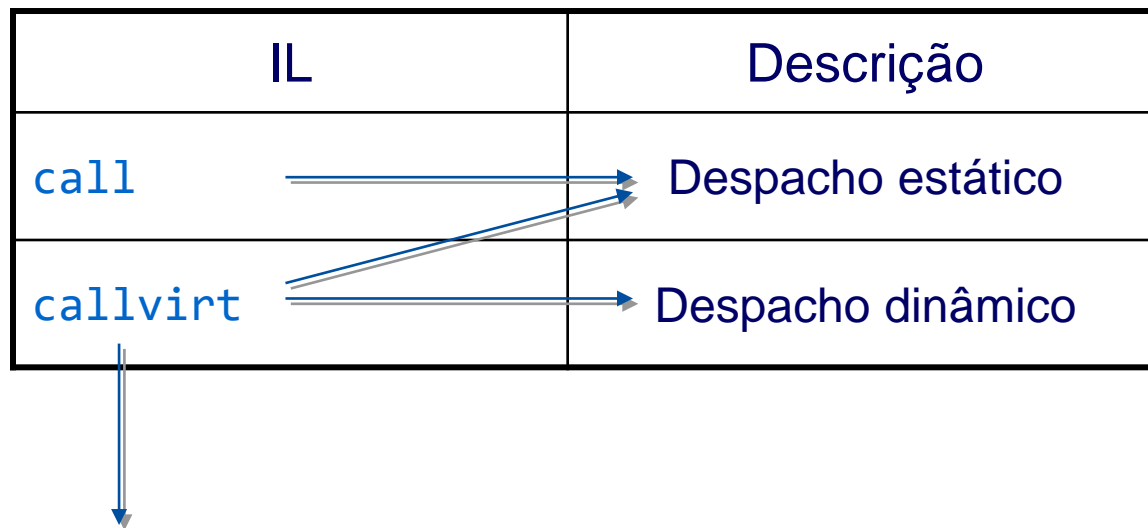


demo

- Invocar o GetType
recorrentemente – Demo 04
-

Invocação de métodos (1)

IL	Descrição
<code>call</code>	Despacho estático
<code>callvirt</code>	Despacho dinâmico



- o `callvirt` verifica adicionalmente se a referência para o objecto é diferente de `null` e lança excepção se for caso disso.
- O compilador de C# gera tipicamente um `callvirt` na chamada a métodos de instância (virtuais ou não virtuais)
 - O que não implica um despacho dinâmico

Invocação de métodos (2)

- P: Em que situações o compilador de C# gera um `call` na chamada a métodos de instância não virtuais?
 - R: Quando tem a certeza de que o `this` é diferente de `null`
 - Exemplo: na invocação de um método de instância dentro dum construtor ou dentro de doutro método de instância

```
class A
{
    public A() {
        M1(); // call
    }
    public void M1() { }
    public void M2()
    { // callvirt
        M1(); // call
    }
    public static void Main()
    {
        A a = new A();
        a.M2();
    }
}
```

demo

Invocação de métodos (3)

- P: É possível invocar um método de instância sobre uma referência a `null`?
 - R: Sim, se não se usar a instância `this` dentro do método (por exemplo, para aceder a um campo de instância)
 - Exemplo:

```
class A
{
    private int a;
    public A() { }
    public void M1()
    { // seria gerado um callvirt pelo Comp. C#
      // mas pode-se usar um call
      // desde que não se use o this
      Console.WriteLine("Olá!");
      this.a = 10; // Excepção!
    }
    public static void Main()
    {
        A a = null;
        a.M1();
    }
}
```

demo

Invocação de métodos (4)

- Como funciona a invocação de métodos não virtuais sobre instâncias de tipos valor?
 - O VES mantém informação de RTTI para o tipo valor
 - Que é usada para aceder à tabela de métodos do tipo valor
 - Note-se que a instância de tipo valor **não contém** uma referência **hType** que permite aceder ao RTTI
 - A referência **hType** só se encontra presente em objectos alojados no heap!
 - A tabela de métodos dum tipo valor contém três tipos de métodos:
 - Métodos estáticos
 - Métodos de instância não virtuais
 - **E métodos virtuais redefinidos no tipo valor** (Ex: redefinição do método **Equals** num tipo valor)

Invocação de métodos (5)

```
// Tipo valor
struct Point
{
    public double x, y;
    public void SetX(double x) {
        this.x = x;
    }
    public static void Main()
    {
        Point p;
        p.x = 10;

        p = new Point(); // Não seria necessário,
        // o comp. C# é que não permite
        // Chama initobj que coloca conteúdo a zeros

        //Console.WriteLine(p.x); // 0

        p.SetX(20);

        //Console.WriteLine(p.x); // 20
    }
}
```

demo

Invocação de métodos (6)

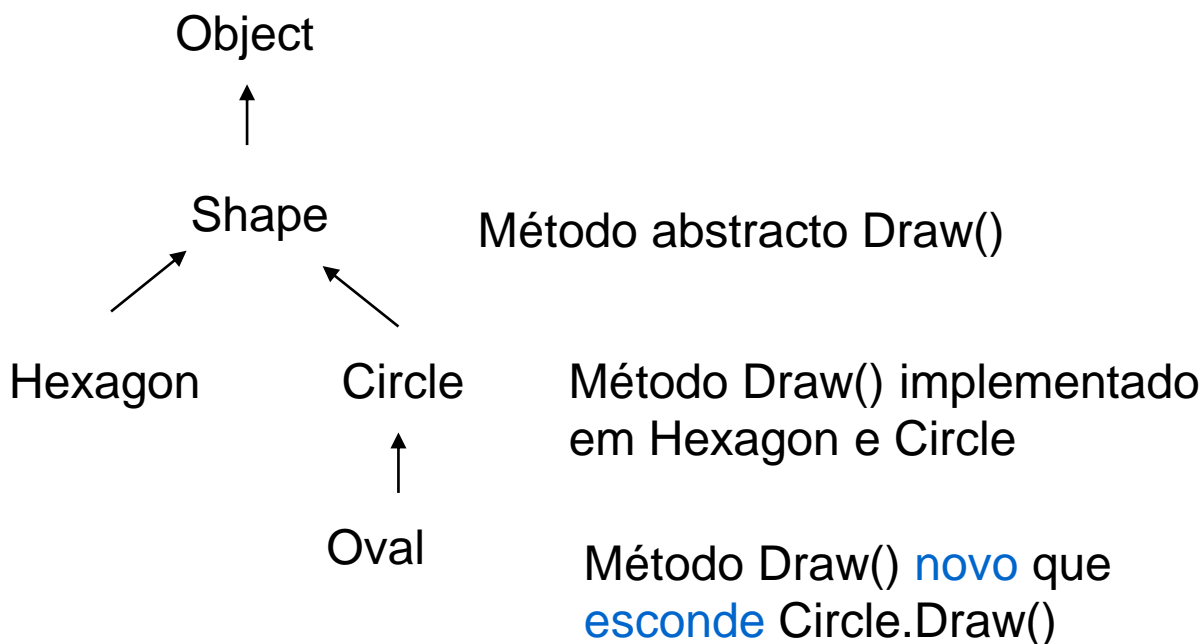
- Invocação de métodos virtuais **redefinidos** num tipo valor
 - O VES invoca o método directamente da tabela presente no RTTI usando um **call – despacho estático – não existe** nenhuma operação de box
 - O VES procede deste modo pois: o tipo valor é **sealed** logo a chamada nunca pode ser polimórfica

```
// Tipo valor
struct Point
{
    public double x, y;
    ...
    public override String ToString() {
        return "(" + this.x + "," + this.y + ")";
    }
    public static void Main()
    {
        Point p = new Point();
        p.x = 20;
        String s = p.ToString();
        Console.WriteLine(s); // 20
    }
}
```

demo

Criar novas versões de membros

- O C# suporta uma funcionalidade que é exactamente o oposto da redefinição de métodos (“overriding”)
 - “Esconder” métodos (“method hiding”)
 - Feito através da *keyword* `new`; pode ser aplicado a outros membros (campos, propriedades, ...)

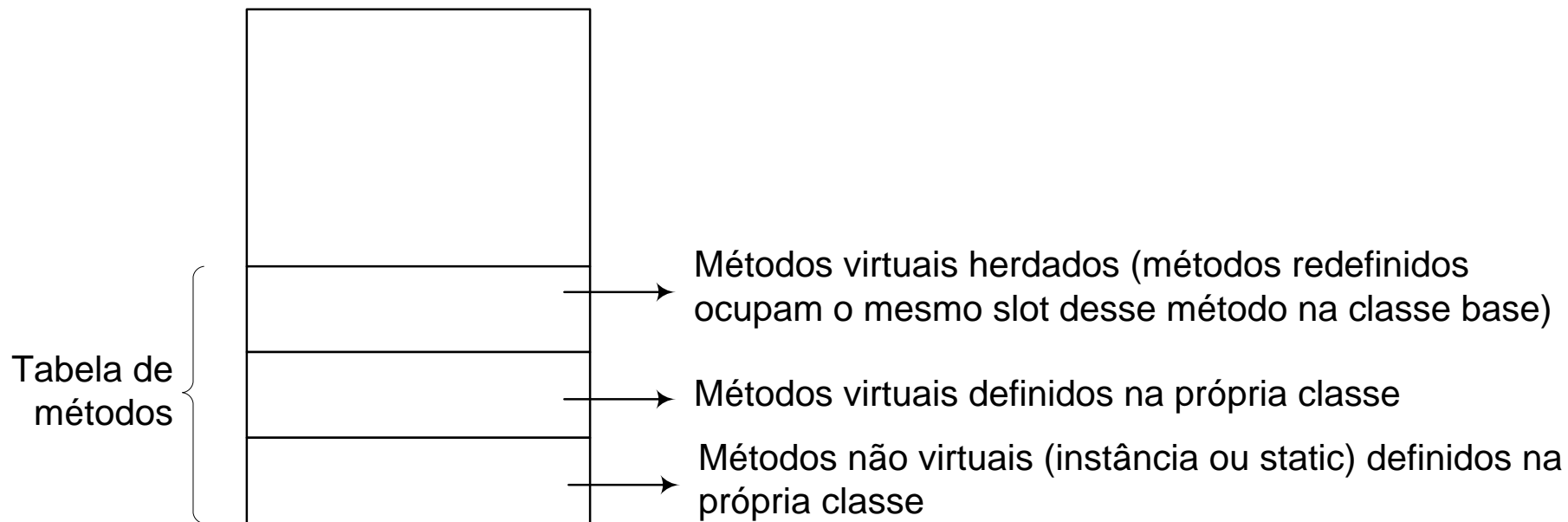


Criar novas versões de membros

```
public class Oval : Circle
{
    public Oval() { }
    // Hide any Draw implementation above me
    public new virtual void Draw()
    {
        // Oval specific drawing algorithm
    }
}
```

Criar novas versões de métodos (1)

COREINFO_CLASS_STRUCT



Criar novas versões de métodos (2)

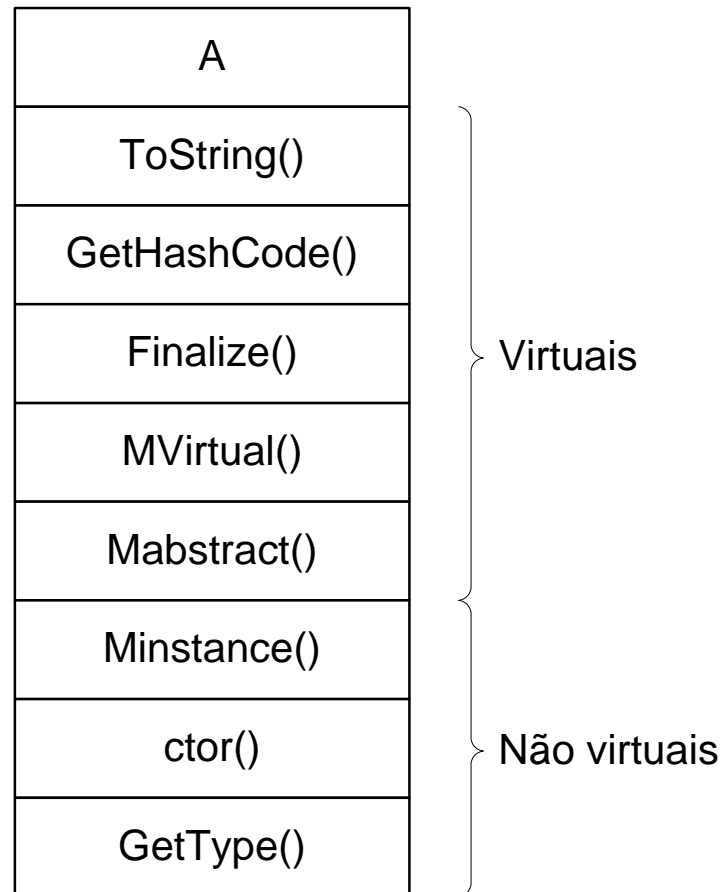
```
public abstract class A {  
    public void MInstance() { ... }  
    public virtual void MVirtual() {  
        ...  
    }  
    public abstract void MAbstract();  
}
```

instance hidebysig
– Gerado por omissão pelo compilador para todos os métodos de instância.

instance hidebysig newslot virtual –
Aparece sempre que o método virtual é declarado pela 1ª vez (não redefine o método da base)

instance hidebysig newslot abstract virtual – Tem que ser implementado pela classe derivada, a não ser que esta também seja abstracta

Criar novas versões de métodos (3)



Criar novas versões de métodos (4)

```
public class B : A {  
    public [new] void MInstance() { ... }  
    public [new, override, new virtual]  
void MVirtual() { ... }  
    public override void MAbstract() {  
... }  
}
```

B tem 2 métodos **MInstance** com a mesma assinatura, não virtuais

- Resolução pelo compilador de C#
- Despacho estático:

b.MInstance()
((A)b).MInstance()

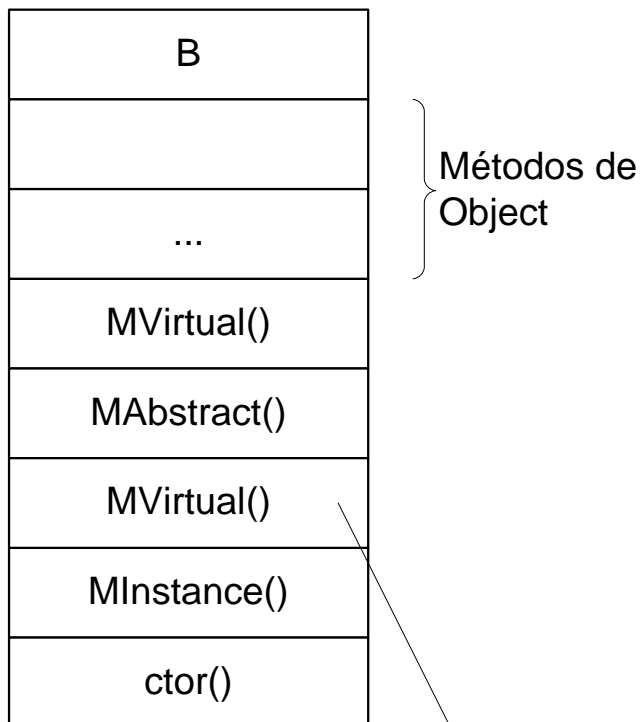
- resolução com base no tipo de referência

override – redefine implementação

new virtual – acrescenta nova implementação virtual, permitindo chamadas polimórficas a partir desta classe. O método da base fica oculto (uso de **newslot** em B)

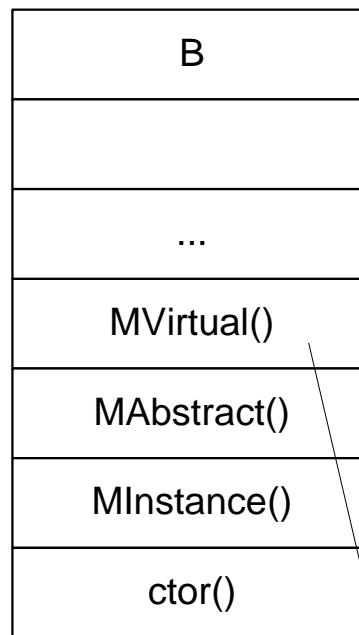
Criar novas versões de métodos (5)

Usando o prefixo **new**



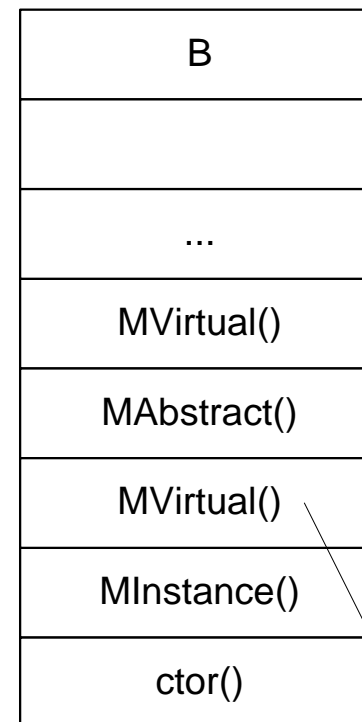
Não é virtual

Com **override**



Nova
implementação

Com **virtual new**



É virtual

Agenda

- Membros de tipos
- Operadores
- Invocação de métodos
- *Explicit Interface Method Invocation (EIMI)*

Implementação explícita de interfaces

- Uma classe pode optar por esconder a implementação de um método da sua “interface” pública.

```
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;  
    }  
}
```

demo

```
public static void Main(){  
    Box myBox = new Box(30.0f, 20.0f);  
    IDimensions myDimensions = (IDimensions) myBox;  
    Console.WriteLine("Length: {0}", myBox.Length());  
    Console.WriteLine("Length: {0}", myDimensions.Length());  
}
```



Implementação explícita de interfaces... aplicabilidade

- Herdar de interfaces que partilham o mesmo nome dos seus membros e dar implementações diferentes.

```
// Declare the "Box" class that implements
// the two interfaces IEnglishDimensions
// and IMetricDimensions:
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;
    }
}
```

```
// Declare the English units interface:
interface IEnglishDimensions {
    float Length();
    float Width();
}
// Declare the metric units interface:
interface IMetricDimensions {
    float Length();
    float Width();
}
```

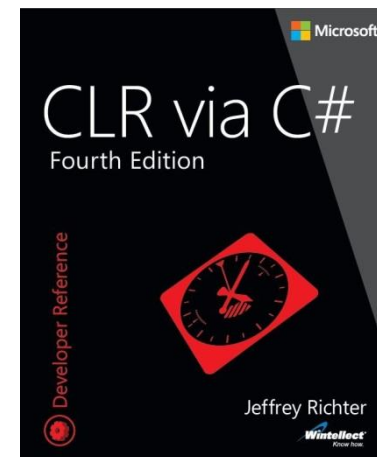
```
public static void Main() {
    // Declare a class instance "myBox":
    Box myBox = new Box(30.0f, 20.0f);
    // Declare an instance of the English units interface:
    IEnglishDimensions eDimensions = (IEnglishDimensions)myBox;
    // Declare an instance of the metric units interface:
    IMetricDimensions mDimensions = (IMetricDimensions)myBox;
    // Print dimensions in English units:
    Console.WriteLine("Length(in): {0}", eDimensions.Length());
    Console.WriteLine("Width (in): {0}", eDimensions.Width());
    // Print dimensions in metric units:
    Console.WriteLine("Length(cm): {0}", mDimensions.Length());
    Console.WriteLine("Width (cm): {0}", mDimensions.Width());
}
```

Implementação explícita de interfaces... resumo

- Designado por **EIMI** – *Explicit Interface Method Implementation*
- O nome do método é prefixado com o nome da interface
- O compilador C# não permite especificar a acessibilidade do método:
 - Método é **private**.
- A única forma de chamar este método é através de uma **referência do tipo da interface**
- O método é sempre **final** e **não** pode ser redefinido.
 - A implementação do método da interface pode ser substituído numa classe derivada desde que esta declare a reimplementação da interface.

Referências

Jeffrey Richter, “CLR via C#, Second Edition”,
Microsoft Press; 4nd edition, 2012



- Don Box, “Essential .NET, Volume I: The Common Language Runtime ”,
Addison-Wesley Professional, 1st edition, 2002

