

---

# CTS via C#

## Parte 1

Centro de Cálculo

Instituto Superior de Engenharia de Lisboa

F. Miguel Carvalho ([mcarvalho@cc.isel.ipl.pt](mailto:mcarvalho@cc.isel.ipl.pt))

# Agenda

---

- “CTS via C#” - Enquadramento
- Tipos Referência
- Tipos Valor
- Conversões

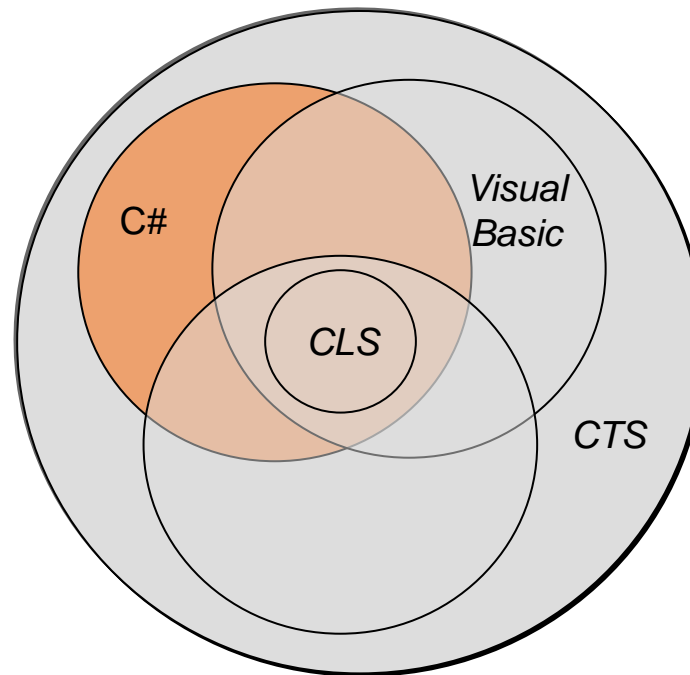
# Agenda

---

- “CTS via C#” - Enquadramento
- Tipos Referência
- Tipos Valor
- Conversões

# "CTS via C#"

- O CTS é a parte do sistema de tipos do CLI:
  - Especificação de como os tipos são definidos e se comportam.



# Classificação de tipos

---

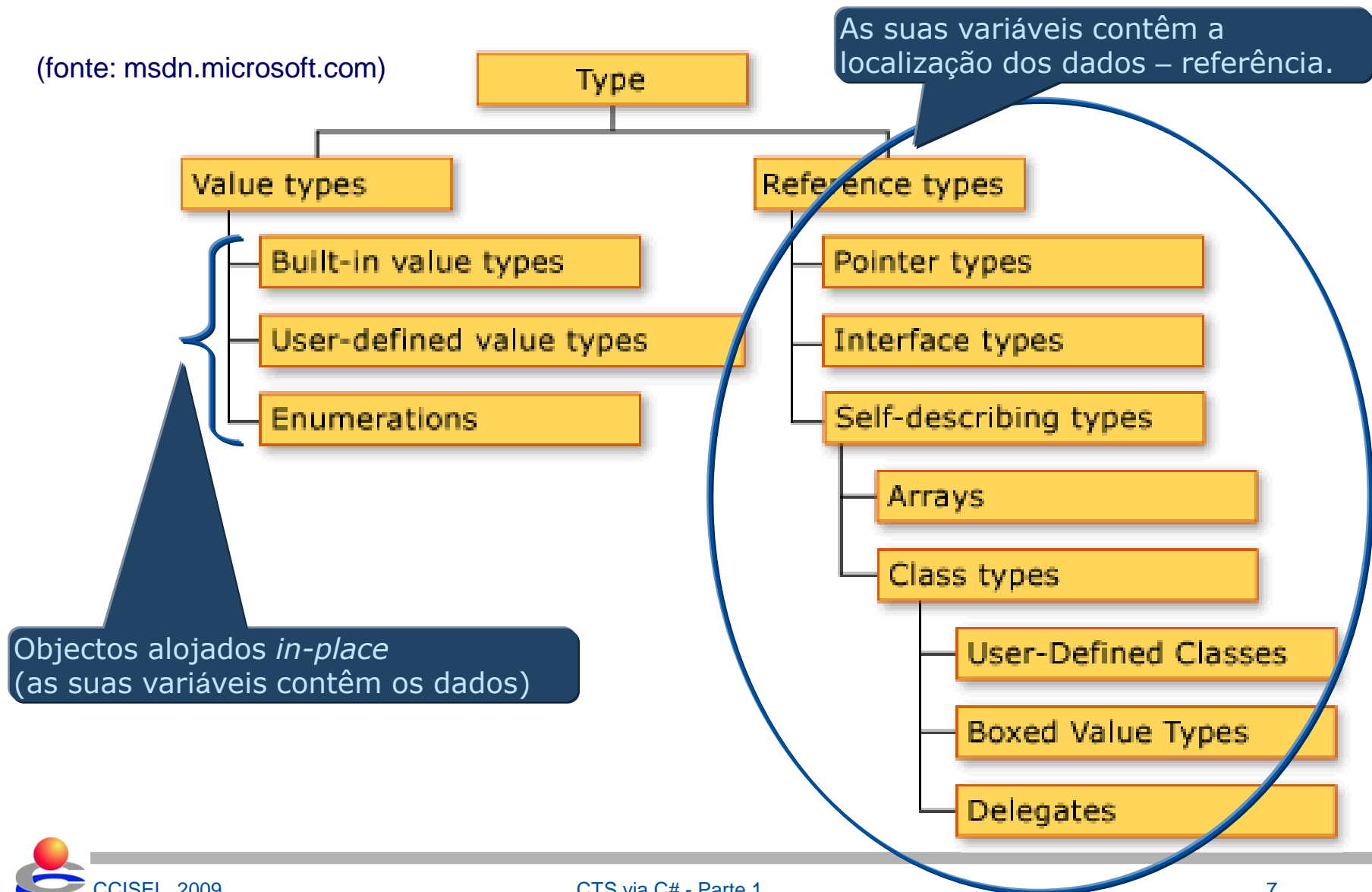
- Tipos Primitivos podem ser identificados como:
  - Tipos básicos:  
nativo; tipo indivisível; unidade de construção;
  - Tipos Intrínsecos ou *built-in*:  
tipo conhecido pela linguagem e com tratamento próprio da linguagem.
- Tipos Compostos
  - Definidos por composição de tipos básicos ou outros tipos compostos.
- Tipos Valor: alojados *in-place*. As suas variáveis contêm os dados.
- Tipos Referência: As suas variáveis contêm a localização dos dados (referência).

# Linguagem C# - Tipos primitivos

C#	BCL	Descrição	R / V
object	System.Object	Base da hierarquia dos tipos <i>Self-Describing</i>	Ref
string	System.String	Unicode string	Ref
bool	System.Boolean	True/false	Val
char	System.Char	Unicode 16-bit char	Val
float	System.Single	IEC 60559:1989 32-bit float	Val
double	System.Double	IEC 60559:1989 64-bit float	Val
sbyte	System.SByte	Signed 8-bit integer	Val
byte	System.Byte	Unsigned 8-bit integer	Val
short	System.Int16	Signed 16-bit integer	Val
ushort	System.UInt16	Unsigned 16-bit integer	Val
int	System.Int32	Signed 32-bit integer	Val
uint	System.UInt32	Unsigned 32-bit integer	Val
long	System.Int64	Signed 64-bit integer	Val
ulong	System.UInt64	Unsigned 64-bit integer	Val
decimal	System.Decimal	A 128-bit high-precision floating-point	Val

# Classificação de tipos

(fonte: msdn.microsoft.com)



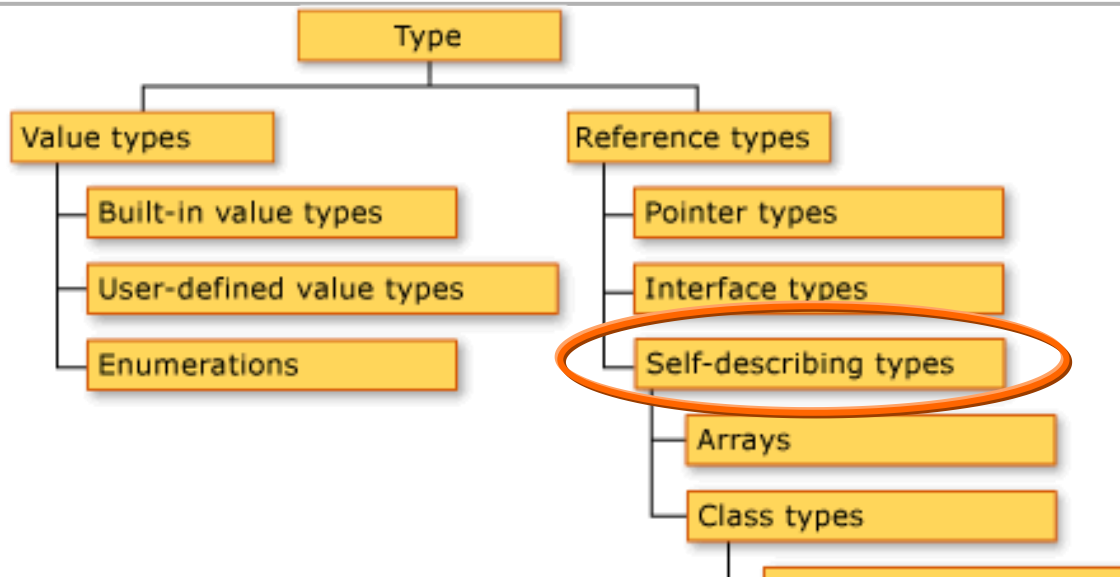
# Agenda

---

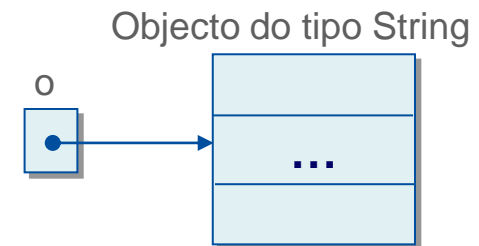
- “CTS via C#” - Enquadramento
- **Tipos Referência**
- Tipos Valor
- Conversões



# Classificação de tipos... “Self-describing Types”



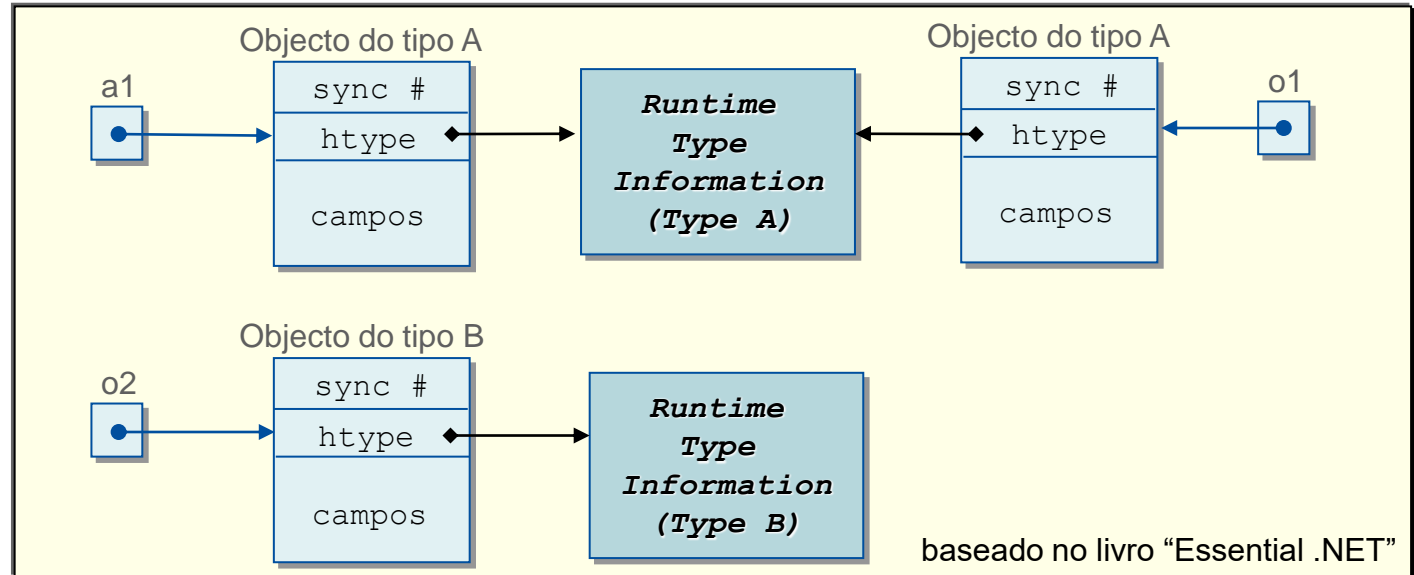
- As instâncias de “Self-describing Types” (designadas objectos)
  - são sempre criadas dinamicamente (alojadas em heap):
    - explicitamente (com o operador new);
    - implicitamente (operação box),
  - a memória que ocupam é reciclada automaticamente (GC);
- Objectos são manipulados através de referências.
- Tipo “real” do objecto pode não coincidir com o tipo da referência:
  - Ex: `Object o = new System.String('A', 10);`



# Informação de tipo em tempo de execução (RTTI)

- A cada objecto é associado um cabeçalho (*object header*: sync# e htype) que descreve o tipo do qual ele é instância.

```
class A{}  
class B{}  
A a1 = new A();  
Object o1 = new A();  
Object o2 = new B();
```

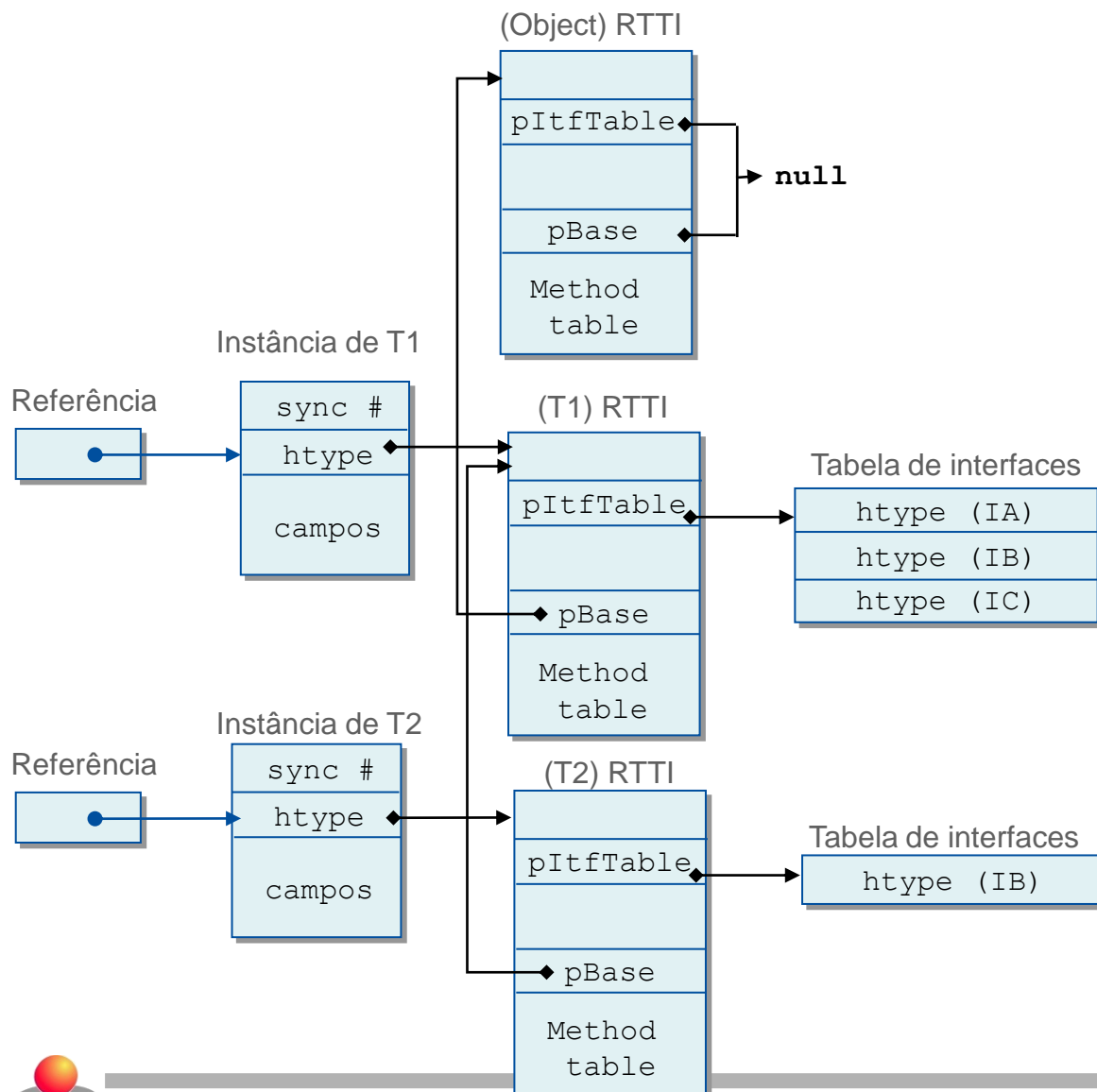


demo

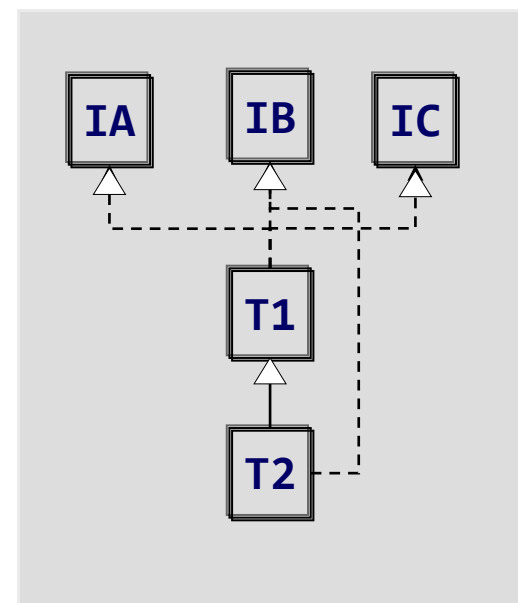
```
class App {  
    static void InstancesOfSameClass(Object o1, Object o2) {  
        if(o1.GetType() == o2.GetType())  
            Console.WriteLine("Instances of same class.");  
        else  
            Console.WriteLine("Instances of different classes.");  
    }  
    static void Main() {  
        ...  
        InstancesOfSameClass(a1, o1);  
        InstancesOfSameClass(o1, o2);  
    } }  
}
```

```
> App.exe  
Instances of same class.  
Instances of different classes.
```

# Informação de tipo em tempo de execução (RTTI)



```
interface IA { }  
interface IB { }  
interface IC { }  
class T1 : IA, IB, IC { }  
class T2 : T1, IB { }
```



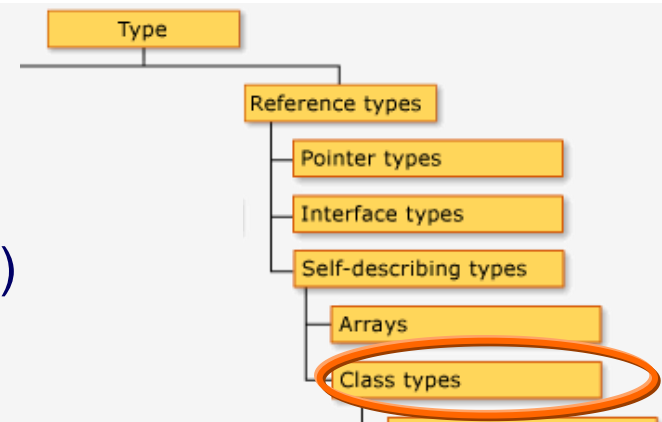
baseado no livro "Essential .NET"

# System.Object

- Base da hierarquia dos “*Self-describing Types*”
  - Contém as operações suportadas por todos eles

```
namespace System {  
  
    public class Object {  
        public virtual bool Equals(object);  
        public virtual int GetHashCode();  
        public virtual string ToString();  
        protected virtual void Finalize();  
        public Type GetType();  
        protected object MemberwiseClone();  
        public static bool Equals(object, object);  
        public static bool ReferenceEquals(object, object);  
        // ...  
    }  
}
```

# Classificação de tipos... “Class Types”

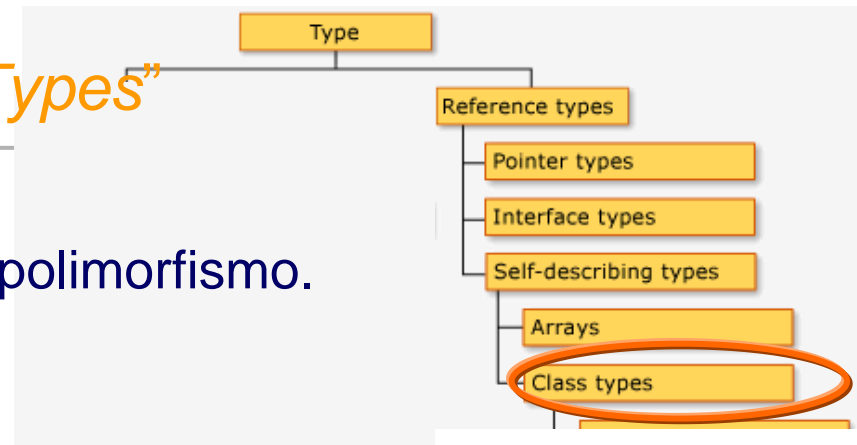


- Construção `class` (para definição de classes)
  - Os membros podem ser:
    - **Dados:** campos
    - **Comportamento:** métodos, propriedades e eventos
    - **Tipos:** *Nested Types*

```
public class C{
    class NestedType{}
    private int Field1;
    private System.EventHandler Field2;
    public C(int aValue){
        Field1 = aValue;
    }
    public event System.EventHandler Event3{
        add{Field2 += value;} //add_Event3 method
        remove{Field2 -= value;} //remove_Event3 method
    }
    public int Property4{
        get{return Field1;} //get_Property4 method
        set{Field1 = value;} //set_Property4 method
    }
}
```

```
.class public auto ansi beforefieldinit C
{
    .class NestedType
    {
        Field1 : private int32
        Field2 : private class [mscorlib]System.EventHandler
        .ctor : void(int32)
        add_Event3 : void(class [mscorlib]System.EventHandler)
        get_Property4 : int32()
        remove_Event3 : void(class [mscorlib]System.EventHandler)
        set_Property4 : void(int32)
        Event3 : [mscorlib]System.EventHandler
        Property4 : instance int32()
    }
}
```

# Classificação de tipos... “Class Types”



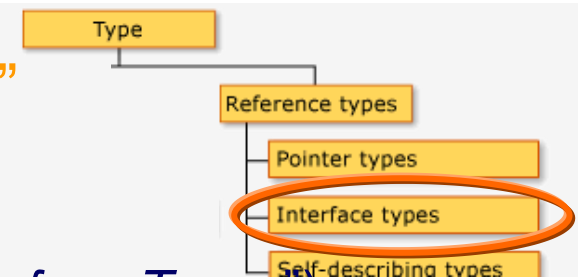
- Suporta encapsulamento, herança e polimorfismo.
- Admite membros de tipo (`static`) e de instância (por omissão).
- Acessibilidades suportadas: `private` (por omissão), `protected` e `public`.
- A relação de herança entre classes designa-se Herança de Implementação
  - Não é admitida utilização múltipla de Herança de Implementação.

# Classificação de tipos... “*Class Types*”

---

- Comportamento: métodos, propriedades e eventos. Suporta:
  - abstractos (abstract)
  - virtuais (virtual):
    - Resolução em tempo de execução, o **tipo do objecto referido** determina o método chamado.
  - Não virtuais (por omissão):
    - Resolução em tempo de compilação, o **tipo da referência** determina o método chamado.

# Classificação de tipos... “Interface Types”



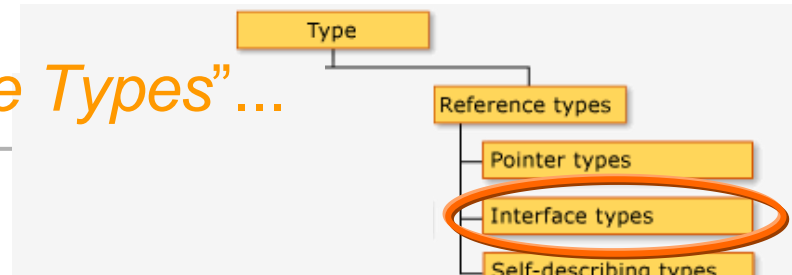
- Construção **interface** (para definição de “Interface Types”)
  - Apenas admite membros de instância: Métodos, propriedades e eventos.
  - Todos os membros (implicitamente) são abstractos (sem implementação);
  - Todos os membros (implicitamente) têm acessibilidade pública.

```
public interface I {  
    void Method1();  
    event System.EventHandler Event2;  
    int Property3 {  
        get; //get_Property3 method  
        set; //set_Property3 method  
    }  
}
```

```
I  
- .class interface public abstract auto ansi  
- Method1 : void()  
- add_Event2 : void(class [mscorlib]System.EventHandler)  
- get_Property3 : int32()  
- remove_Event2 : void(class [mscorlib]System.EventHandler)  
- set_Property3 : void(int32)  
- Event2 : [mscorlib]System.EventHandler  
- Property3 : instance int32()
```



# Classificação de tipos... “Interface Types”...



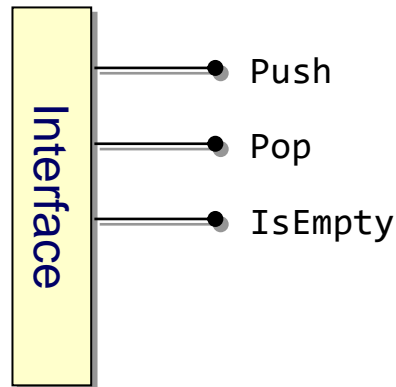
- Para especificação de contratos, isto é, conjunto de operações suportadas.
- A relação de herança entre classes e interfaces, e entre interfaces e interfaces designa-se Herança de Interface
  - É admitida utilização múltipla de Herança de Interface
- Por convenção, o nome das interfaces começa pelo carácter ‘I’
  - Exemplos: **ICloneable**, **IEnumerable**

# Interfaces

## Implementação

```
class Stack : IntStack {
    private int[] stk;
    private int ptr;
    public Stack(int dim) {
        stk = new int[dim];
    }
    // Pre: Stack is not full
    public void Push(int v)
    { stk[ptr++] = v; }
    // Pre: Stack is not empty
    public int Pop()
    { return stk[--ptr]; }
    public bool IsEmpty()
    { return ptr == 0; }
```

IntStack



```
interface IntStack {
    void Push(int v);
    int Pop();
    bool IsEmpty();
}
```

## Utilização

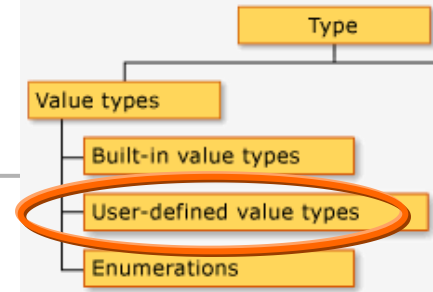
```
class Program {
    public void static Main() {
        Stack s = new Stack(10);
        // Adicionar elementos
        for (int i = 0; i < 10; ++i)
            s.Push(2);
        // Remover todos
        while (!s.IsEmpty())
            Console.WriteLine(s.Pop());
    }
}
```

# Agenda

---

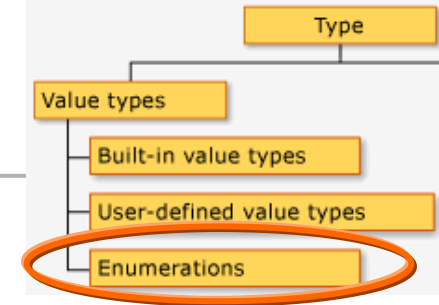
- “CTS via C#” - Enquadramento
- Tipos Referência
- **Tipos Valor**
- Conversões

# User-defined value types



- Construção **struct** (para definição de Tipos Valor)
  - Os **membros** podem ser os mesmos de uma **class**:
    - Dados: campos
    - Comportamento: métodos, propriedades e eventos
    - Tipos: *Nested Types*
- Esta construção apenas dá suporte ao encapsulamento
- O Tipo Valor definido não admite tipos derivados
- Admite membros de tipo (**static**) e de instância (por omissão)
- Acessibilidades suportadas: **private** (por omissão) e **public**

# Enumerados



- Em C#

```
public enum ConsoleColor {  
    Black,  
    ...  
    White  
}
```

- Em CIL

```
class public auto ansi serializable sealed ConsoleColor extends System.Enum {  
    .field public specialname rtspecialname int32 value__  
    .field public static literal valuetype System.ConsoleColor Black = int32(0)  
    ...  
    .field public static literal valuetype System.ConsoleColor White = int32(15)  
}
```

- *Bit flags*

```
[Flags]  
public enum FileAttributes {  
    Archive = 0x20,  
    ...  
    Temporary = 0x100  
}
```

# Unificação do sistema de tipos - *Boxing e Unboxing*

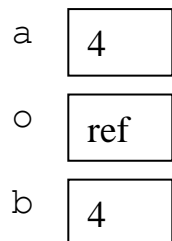
C#

```
int a = 4;  
object o = a;  
int b = (int) o;
```

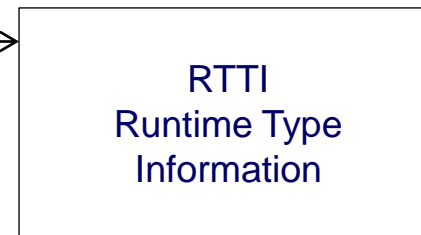
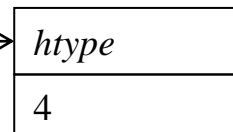
IL

```
{  
  ldc.i4.4  
  stloc.0  
  ldloc.0  
  box [mscorlib]System.Int32  
  stloc.1  
  ldloc.1  
  unbox.any [mscorlib]System.Int32  
  stloc.2  
}
```

Stack



Heap



---

# exercício

Box e unbox

---

# Agenda

---

- “CTS via C#” - Enquadramento
- Tipos Referência
- Tipos Valor
- Conversões



# Conversão entre tipos referência (1)

---

- Localização e objectos (referidos pelas localizações) podem ser de tipos diferentes
  - Objectos são sempre de tipos exactos
  - Variáveis podem ser de tipo exacto ou de outros tipos da hierarquia
- *Casting*: guardar numa variável de tipo T um objecto cujo tipo é “compatível”
  - É normalmente resolvido em tempo de compilação, mas pode ter de ser adiado até à fase de execução
  - Na linguagem C#: operadores *as*, *is* e *casting*.

## Conversão entre tipos referência (2)

- *Casting*: conversão entre tipos referência
- Coerção: conversão de instâncias de tipo valor primitivas (intrínsecas ao VES)
- *Casting* ≠ coerção

```
// Coerção  
int i;  
long l;  
l = i;  
i = (int)l;
```

```
class A {}  
class B : A {} // B deriva de A  
  
// Casting  
A a;  
B b = new B();  
a = b; // upcast implícito  
B b1 = (B)a; // downcast explícito
```


demo

## Conversão entre tipos referência (3)

- Operadores do C#:

- Operador *casting*

- Operador `is`:  $(a \text{ is } B) \begin{cases} \text{true} \\ \text{false} \end{cases}$

- Operador `as`:  $(a \text{ as } B) \rightarrow \text{valor} \begin{cases} a \\ \text{null} \end{cases}$   


Referência      Tipo

- Exemplo: No código `b = a as B`, se `a` referir um objecto do tipo `B` então `b` passa a referir esse objecto;

- Caso contrário, `b` passa a referir `null`.

## Conversão entre tipos referência (4)

```
public class Employee { }
public class Manager : Employee { }
public sealed class Program {
    public static void Main() {
        Manager m = new Manager();
        ProcessEmployee(m);
    }
    public static void ProcessEmployee(Employee e) {
        if (e is Manager) {
            Manager m = (Manager) e;
            // ...
        }
    }
}
```

Devolve **true** se a instância referida por **e** for compatível com **Manager**;  
Devolve **false** caso contrário

Operador de **casting**:  
Resulta em **ClassCastException** se a instância referida por **e** não for compatível com o tipo **Manager**

demo

## Conversão entre tipos referência (5)

---

- Operadores `is` e `as` são implementados pelo *opcode* `isinst` do IL
- Processo de verificação da compatibilidade com um tipo:
  - (1) Verifica se esse tipo é igual ao tipo a testar
    - Se for igual, termina
    - Se não vai consultar o `hType` do tipo base
    - Faz recorrentemente até:
      - Chegar a `System.Object`
      - Encontrar a classe igual ao tipo a testar
  - (2) Percorre a tabela de interfaces referida por `pItfTable`
  - (3) Se não encontrar um tipo igual retorna `null`
- Este processo pode ser dispendioso se a estrutura da classe for complexa (hierarquia complexa, várias interfaces)
- O operador `castclass` (*casting*) faz o mesmo processo de verificação mas *lança exceção em caso de incompatibilidade*

# Conversão entre tipos numéricos (coerção) - 1

- Coerção: guardar uma instância de tipo valor *built-in* numa variável cujo tipo não é compatível com o da instância
  - Pode resultar em alterações de representação e de tipo
  - Na linguagem C#:
    - Coerção com alargamento (*widening*): implícita
    - Coerção com diminuição de resolução (*narrowing*): explícita
    - *Narrowing* trunca os *bits* mais significativos
    - *Narrowing* pode lançar exceção (*overflow*)

```
int i = 256;  
byte b = 0;  
checked {  
    b = (byte)i;  
}
```

System.OverflowException:  
Arithmetic operation  
resulted in an overflow

## Conversão entre tipos numéricos (coerção) - 2

```
static void Main() {
```

```
    Int32 i32 = 1;
```

```
    Int64 i64 = 1;
```

```
    i64 = i32;
```

```
    i32 = (Int32) i64;
```

```
    i64 = Int64.MaxValue;
```

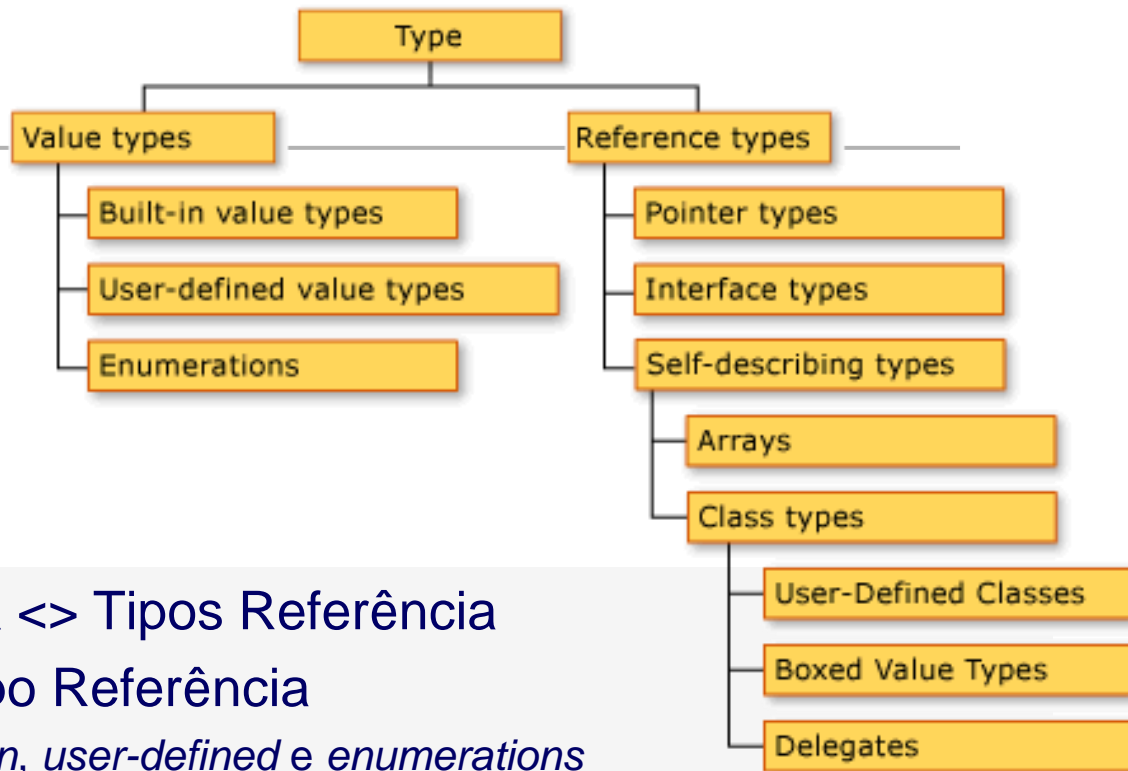
```
    i32 = (Int32) i64;
```

```
    i32 = checked((Int32) i64);
```

```
}
```

{	IL_0000:	ldc.i4.1
	IL_0001:	stloc.0
{	IL_0002:	ldc.i4.1
	IL_0003:	conv.i8
	IL_0004:	stloc.1
{	IL_0005:	ldloc.0
	IL_0006:	conv.i8
	IL_0007:	stloc.1
{	IL_0008:	ldloc.1
	IL_0009:	conv.i4
	IL_000a:	stloc.0
	IL_000b:	ldc.i8 0x7fffffffffffffffffff
	IL_0014:	stloc.1
{	IL_0015:	ldloc.1
	IL_0016:	conv.i4
	IL_0017:	stloc.0
	IL_0018:	ldloc.1
	IL_0019:	conv.ovf.i4
	IL_001a:	stloc.0
	IL_001b:	ret

# Conversões (resumo)



- Casting: Tipos Referência <> Tipos Referência
- *Boxing*: Tipos Valor → Tipo Referência  
Inclui todos os tipos valor: *built-in*, *user-defined* e *enumerations*  
Os Tipos referência são apenas: interfaces, tipo `System.ValueType` e tipo `System.Object`.
- Unboxing: Tipos Valor ← *Boxed Value Types*
- Impossível: Tipos Valor *User-defined* <> Tipos Valor User-defined
- Coerção: Tipos básicos <> Tipos básicos  
“Built-in value types”



# Unificação do sistema de tipos (resumo)

---

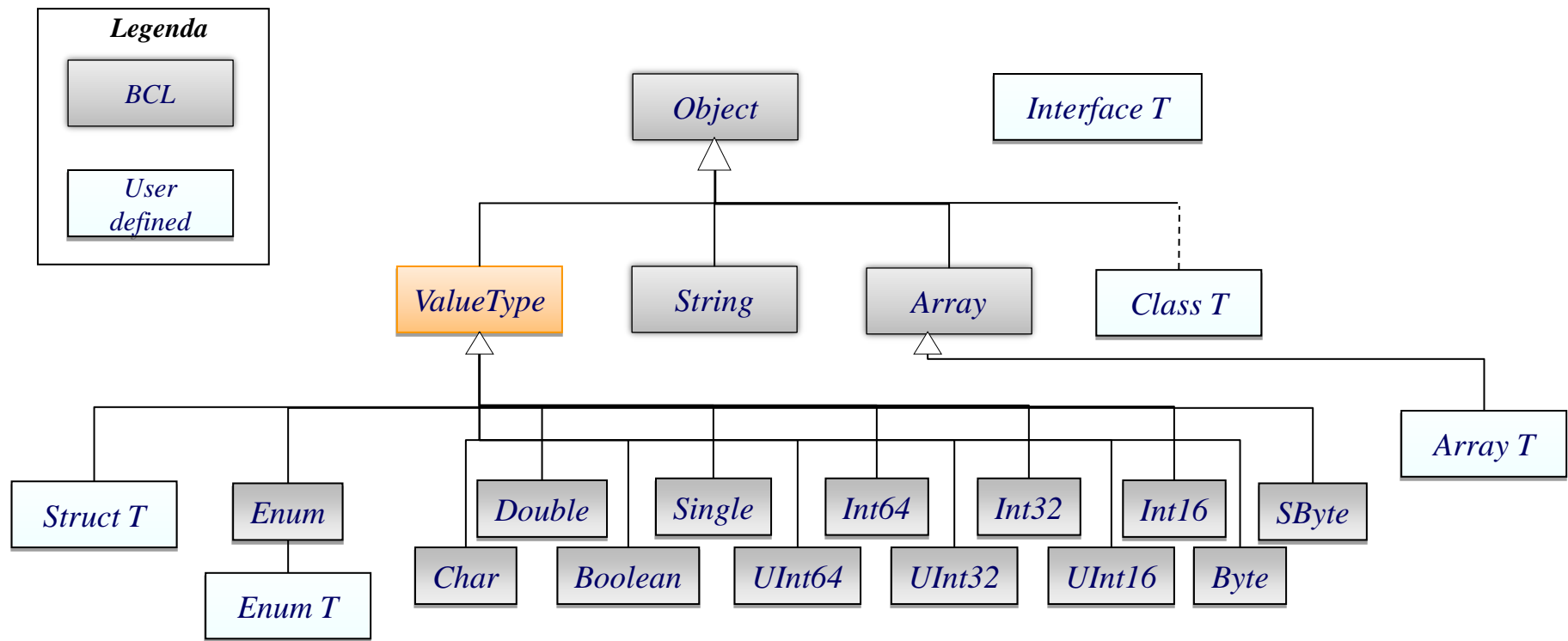
- As variáveis
  - de Tipos Valor contêm os dados
  - de Tipos Referência contêm a localização dos dados
- As instâncias de “*Self-describing Types*” (designadas *objectos*)
  - são sempre criadas dinamicamente (em *heap*)
    - explicitamente (com o operador **new**)
    - implicitamente (operação *box*)
  - a memória que ocupam é reciclada automaticamente (GC)

# Unificação do sistema de tipos (resumo)

---

- é sempre possível determinar o seu tipo exacto
  - em tempo de execução todos os objectos incluem ponteiro para o descritor do tipo a que pertencem
- A cada Tipo Valor corresponde um “*Boxed Value Type*”
  - Suporte para conversão entre Tipos Valor e Tipos Referência (*box* e *unbox*)

# Linguagem C# - Hierarquia de tipos (BCL) revisitada



# 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

