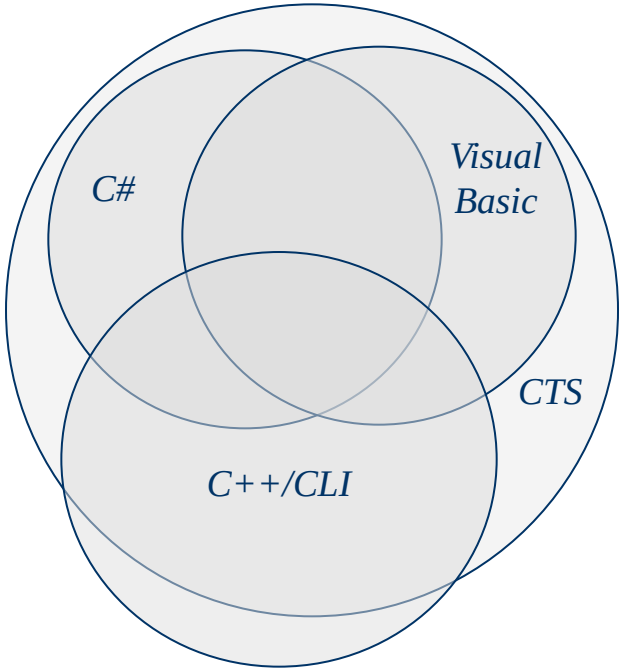
Common Type System (CTS)

Notas para a disciplina de
Ambientes Virtuais de Execução
Semestre de Inverno, 10/11

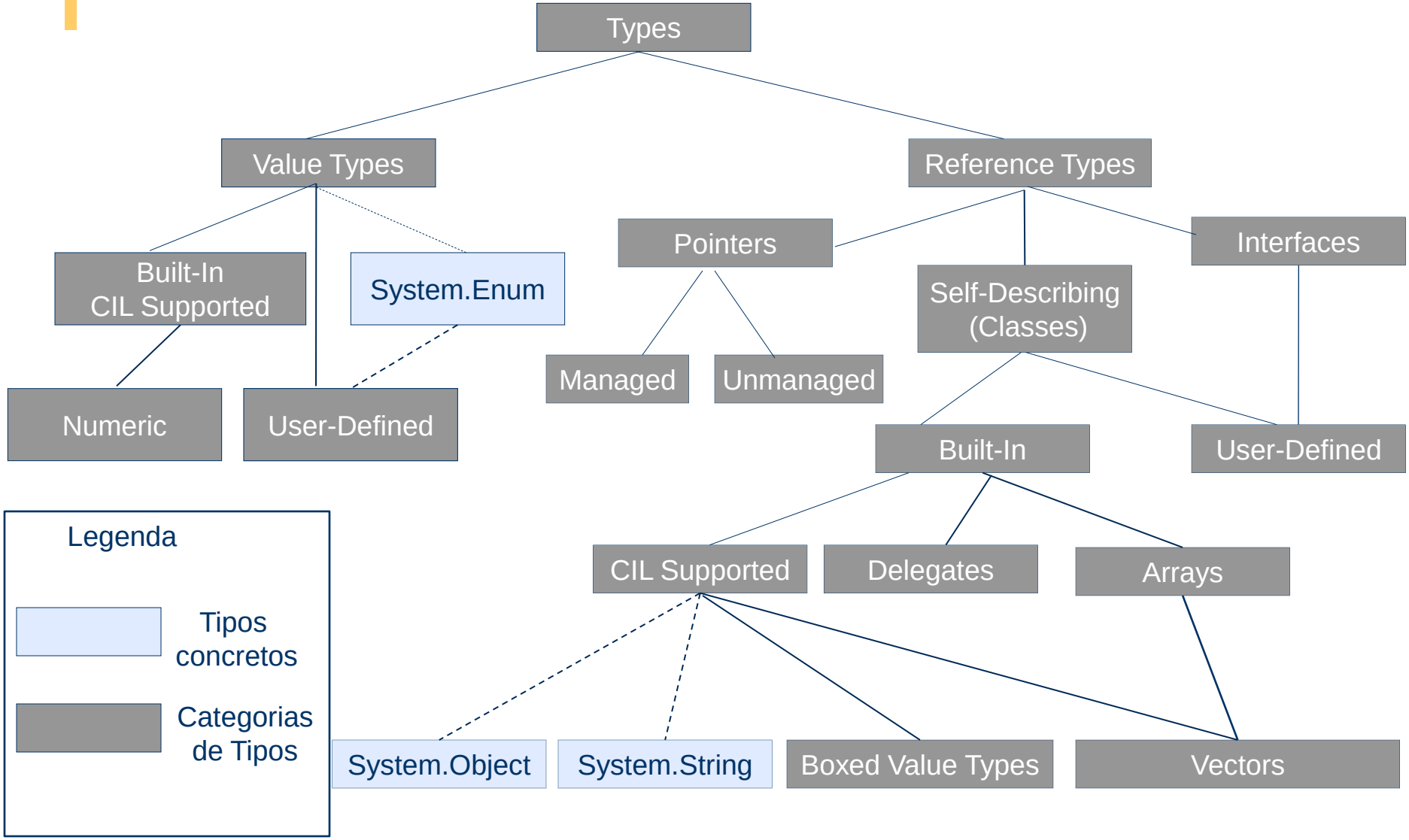
Common Type System (CTS)

- Sistemade tipos (orientado aos objectos) que permite representar tipos de acordo com os modelos de tipos de várias linguagens.
  - Idealmente deveria constituir a UNIÃO dos sistemas de tipos das linguagens a suportar. Não se conhecendo à priori todas as linguagens a suportar é um objectivo difícil…

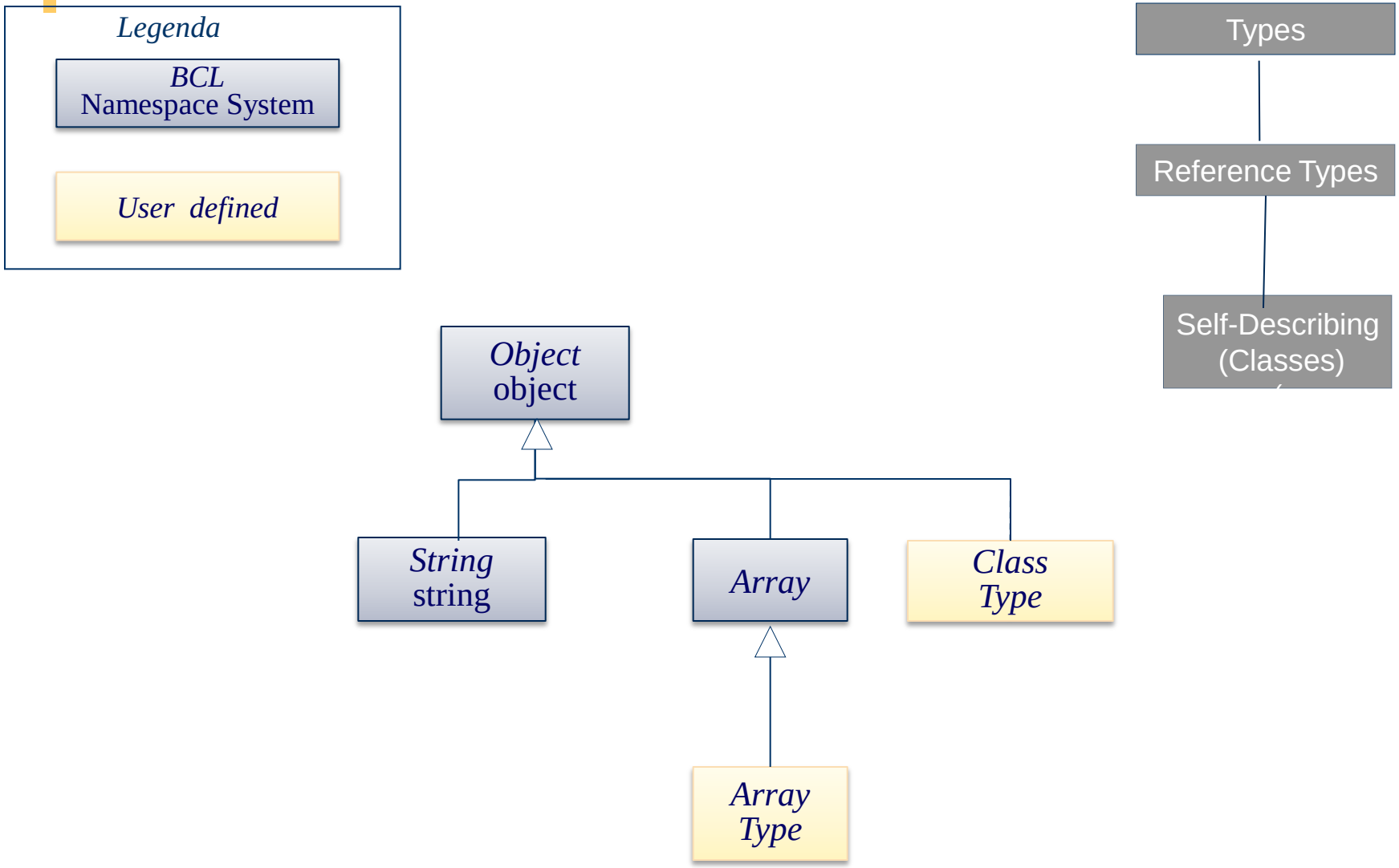
- *Common Type System* (CTS) define:
  - categorias de tipos
    - tipos valor e tipos referência
  - hierarquia de classes
  - conjunto de tipos *built-in*
  - construção e definição de novos tipos e respectivos membros
  - tipos genéricos



Common Type System – Metamodelo, categorias de tipos no CTS

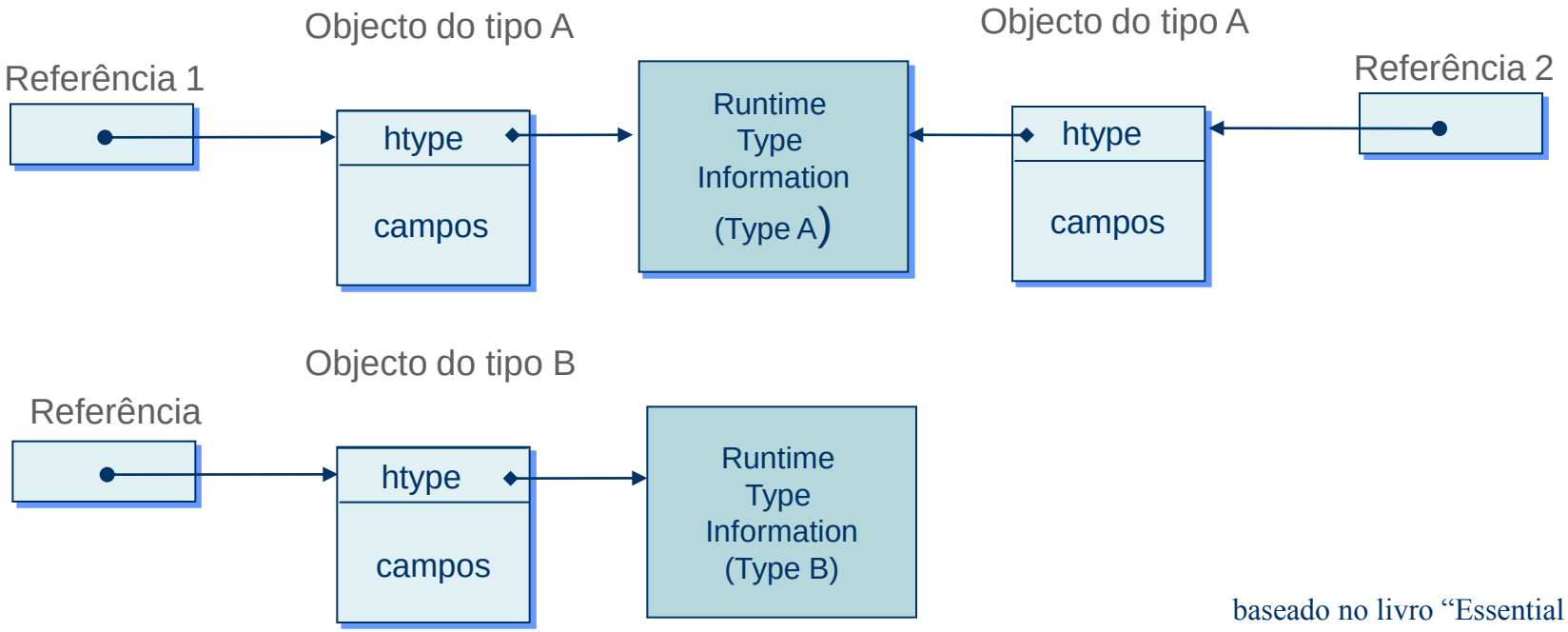


Excerto do modelo de tipos - classes



Informação de tipo em tempo de execução (RTTI)

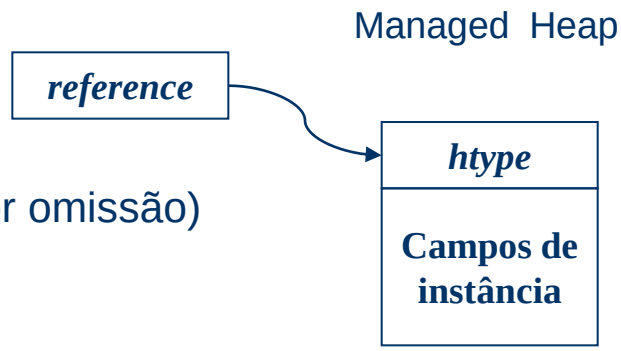
- Objectos são manipulados através de referências
  - Tipo “real” do objecto pode não coincidir com o tipo da referência
    - .e.g. `Object o = new System.String('A', 10);`
- A cada objecto é associada uma estrutura de dados que descreve o tipo do qual ele é instância (object header)



baseado no livro “Essential .NET”

Linguagem C# - construção *de tipos (classes)*

- Keyword `class` (para definição de classes)



  - Suporta encapsulamento, herança e polimorfismo
  - Admite membros de tipo (**`static`**) e de instância (por omissão)
  - Os membros podem definir:
    - Dados: campos
    - Comportamento: métodos, propriedades e eventos
      - virtuais (**`virtual`**) ou abstractos (**`abstract`**)
    - Tipos: *Nested Types* (sempre membros estáticos)
    - Acessibilidade (dos membros): **`private`** (por omissão), **`family`** e **`public`**
  - Acessibilidade (do tipo): **`private`**(por omissão: interno ao *assembly* onde está definido), e **`public`**
  - Semântica de cópia: cópia da referência
  - Herda (de `System.Object`) uma implementação de `Equals` que compara identidade
- A relação de herança entre classes designa-se Herança de Implementação
  - Não é admitida utilização múltipla de Herança de Implementação

Linguagem C# - System.Object

- Base da hierarquia dos "*Self-describing Types*"
  - Contém as operações comuns a todos eles

```
namespace System {

    public class Object {
        public Type GetType();
        public virtual bool Equals(object);
        public virtual int GetHashCode();
        public virtual string ToString();

        protected virtual object MemberwiseClone();
        protected virtual void Finalize();

        public static bool Equals(object, object);
        public static bool ReferenceEquals(object, object);
    }
}
```

Métodos de instância de Object

Virtual publico *Boolean Equals(Object o)*	Por omissão compara a identidade do objecto invocado com o objecto passado como parâmetro. Pode ser redefinido para comparar conteúdo em vez de identidade
Virtual publico *Int GetHashCode()*	Retorna a chave (*hash code*) a usar na inserção do objecto numa tabela de *hash*. Terá de ser síncrono com o método `Equals`, isto é se dois objectos são iguais, terão de ter o mesmo *hash code*.
Virtual publico *ToString()*	Por omissão retorna o nome completo do tipo - `this.GetType().FullName` É comum ser redefinido. Por exemplo na classe `System.Boolean` retorna uma representação (`true` ou `false`) do valor da instância. É comum depender da cultura corrente.
publico *Type GetType()*	Retorna um objecto de um tipo derivado de `Type` que identifica o tipo do objecto invocado
Virtual protegido *Void Finalize()*	Usado no processo de recolha automática de memória
protegido *Object MemberwiseClone()*	Retorna um *clone* do objecto invocado por *shallow copy* (cópia não recursiva)

Hierarquia de Classes do tipo de um objecto

```
class Ancestors {

    public static void Show(object o) {
        Type t = o.getType();
        while(true) {
            Console.WriteLine(t.FullName);
            if (t== typeof(System.Object)) break;
            t = t.BaseType;
        }
    }
}
```

Igualdade e identidade

- Identidade e Igualdade são operações sobre valores definidas no CTS como relações de equivalência:
  - Reflexiva - V op V é verdade
  - Simétrica – se V1 op V 2 é verdade então V2 op V1 é verdade
  - Transitiva – se V1 op V2 é verdade e V2 op V3 é verdade então V1 op V3 também é verdade
  - A identidade implica igualdade mas o inverso não é verdadeiro

Equivalência e Identidade em Object

```
class Object {
    /* igualdade em object – compara identidade! */
    public virtual Boolean Equals(Object obj) {
        return Object.ReferenceEquals(this, obj);
    }

    /* identidade em object */
    public static bool ReferenceEquals(object o1, object o2) {
        return o1 == o2;
    }

    public static bool Equals(object o1, object o2)  {
        if (o1 == o2) return true;
        if ((o1 == null) || (o2 == null))
            return false;
        return o1.Equals(o2);
    }

}
```

Redefinido sempre em associação com `GetHashCode`: Dados `a` e `b` então se `a.Equals(b) == true => a.GetHashCode == b.GetHashCode()` (O inverso não é verdade).

Conversões implícitas e explícitas entre tipos (casting) – Parte 1

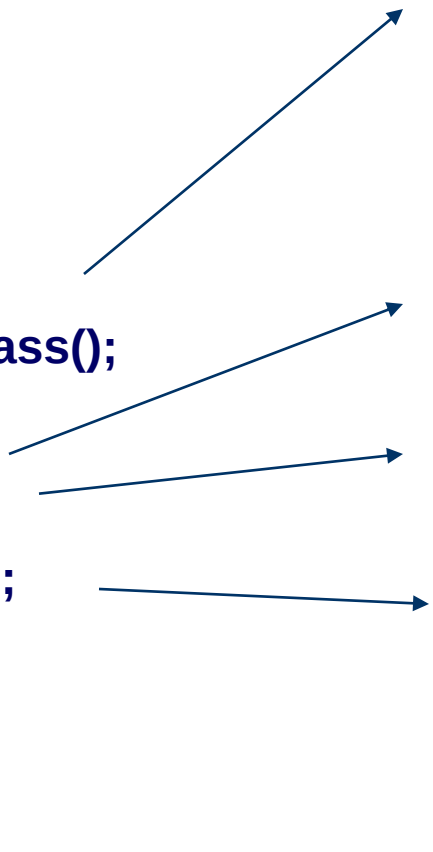
- Localização e objectos (referidos pelas localizações) podem ser de tipos diferentes
  - Objectos são sempre de tipos exactos
  - Variáveis podem ser de tipo exactos (classes não abstractas) ou incompletos (classes abstractas e interfaces)

- Casting: Guardar numa variável de tipo T um objecto cujo tipo é “compatível”

  - Resolvido em tempo de compilação (*upcast*), mas pode ter de ser adiado até à fase de execução (downcast).
  - Na linguagem C#: operadores **`as`** e **`is`** e **`(`***Base***`)`**

Conversões implícitas e explícitas entre tipos (casting) – Parte 2

```
using System;
class Aclass {
    public int i=0;
}

class MainClass
{
    static void Main(){
      AClass a = new AClass();
      Object o = a;
      a = (AClass) o;
      a = o as AClass;
      bool b = o is AClass;
    }
}
```

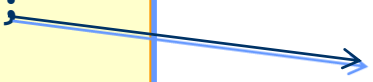


```
IL_0000:  newobj     instance void
      AClass::.ctor()
IL_0005:  stloc.0
IL_0006:  ldloc.0
IL_0007:  stloc.1
IL_0008:  ldloc.1
IL_0009:  castclass  AClass
IL_000e:  stloc.0
IL_000f:  ldloc.1
IL_0010:  isinst     AClass
IL_0015:  stloc.0
IL_0016:  ldloc.1
IL_0017:  isinst     AClass
IL_001c:  ldnull
IL_001d:  cgt.un
IL_001f:   stloc.2
IL_0020:  ret
```

Conversão entre tipos numéricos (coerção) – Parte 1

- Coerção: Guardar uma instância de tipo valor cujo tipo não é compatível com o da variável
  - Pode resultar em alterações de valor
  - Na linguagem C#:
    - Coerção com alargamento (*widening*): implícita
    - Coerção com diminuição de resolução (*narrowing*): explícita
    - *Narrowing* pode lançar excepção (*overflow*)

```
int i = 1024;
byte b = 0;
Checked {
    b = (byte)i;
}
```



System.OverflowException:
Arithmetic operation
resulted in an overflow

Conversão entre tipos numéricos (coerção) – Parte 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);
}
```

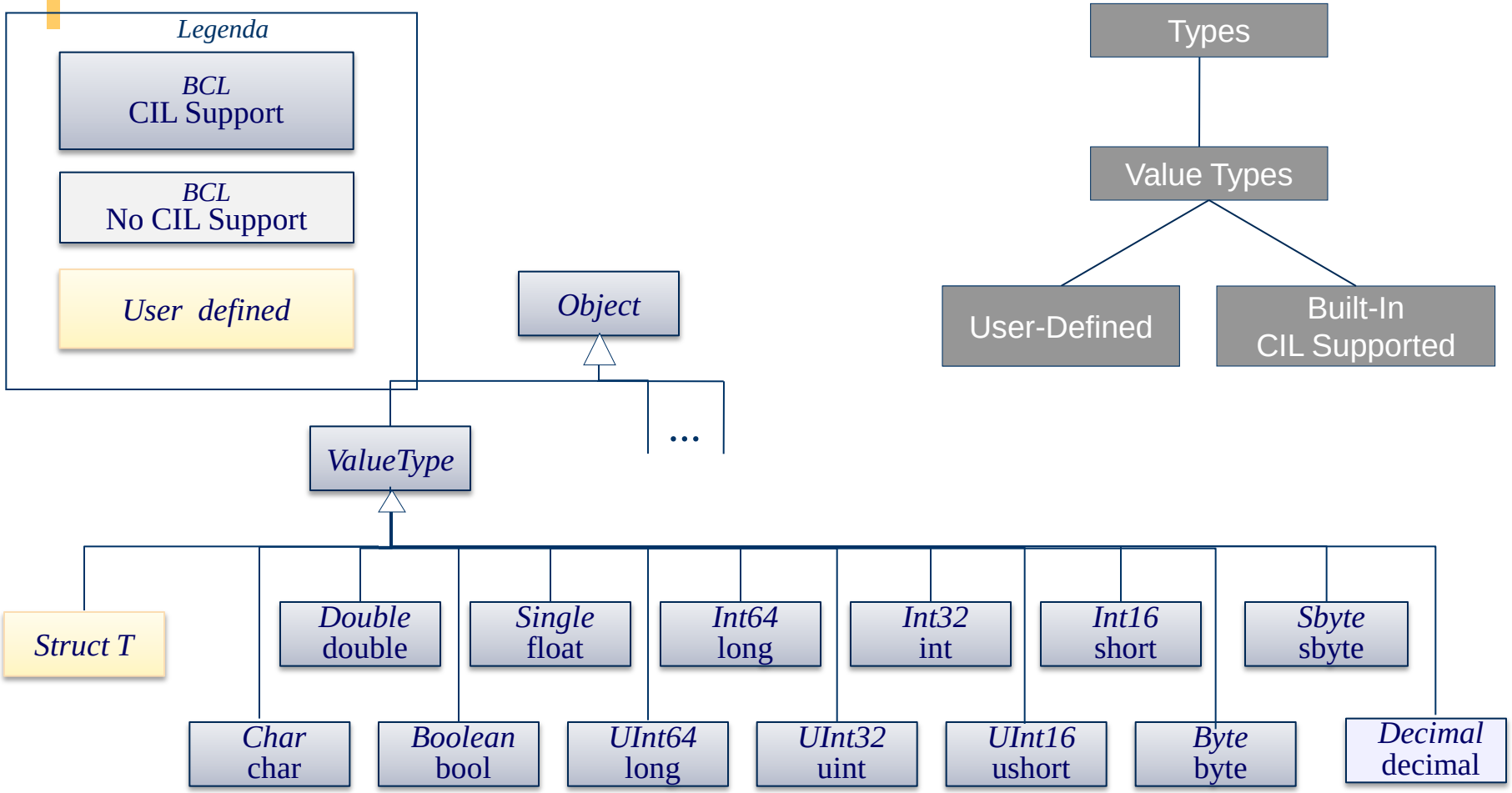
```
IL_0000:  ldc.i4.1
IL_0001:  stloc.0
IL_0002:  ldc.i4.1
IL_0003:  conv.i8
IL_0004:  stloc.1
IL_0005:  ldloc.0
IL_0006:  conv.i8
IL_0007:  stloc.1
IL_0008:  ldloc.1
IL_0009:  conv.i4
IL_000a:  stloc.0
IL_000b:  ldc.i8    0x7fffffffffffffff
IL_0014:  stloc.1
IL_0015:  ldloc.1
IL_0016:  conv.i4
IL_0017:  stloc.0
IL_0018:  ldloc.1
IL_0019:  conv.ovf.i4
IL_001a:  stloc.0
IL_001b:  ret
```

Exemplo de classe : Stack implementado com lista ligada

```
.class public auto Stack
extends [mscorlib]System.Object
```

```
public class Stack {
    private class Entry {
       public object val;
       public Entry next;
       public Entry(object val, Entry next) {
            this.val=val; this.next=next;
       }
    }
    private Entry head;
    public void Push(object o) { head = new Entry(o, head);  }

    public object Pop() {
       if (head == null) throw new InvalidOperationException("Empty Stack");
       object val = head.val;
       head = head.next;
       return val;
    }
    public bool Empty()  { return head==null; }
}
```

Excerto do modelo de tipos (Tipos valor)



Linguagem C# - Tipos intrínsecos (ou *built-in*)

BCL	C#	Descrição	Ref / Value
System.Object	object	Base da hierarquia dos tipos *Self-describing*	Ref
System.String	string	Unicode string	Ref
System.Boolean	bool	True/false	Val
System.Char	char	Unicode 16-bit char	Val
System.Single	float	IEC 60559:1989 32-bit float	Val
System.Double	double	IEC 60559:1989 64-bit float	Val
System.SByte	sbyte	Signed 8-bit integer	Val
System.Byte	byte	Unsigned 8-bit integer	Val
System.Int16	short	Signed 16-bit integer	Val
System.UInt16	ushort	Unsigned 16-bit integer	Val
System.Int32	int	Signed 32-bit integer	Val
System.UInt32	uint	Unsigned 32-bit integer	Val
System.Int64	long	Signed 64-bit integer	Val
System.UInt64	ulong	Unsigned 64-bit integer	Val
System.Decimal	decimal	Valores decimais com precisão estendida	Val

Linguagem C# - construção *de tipos (Tipos Valor)*

- Keyword `struct` (para definição de Tipos Valor)
  - Esta construção apenas dá suporte ao encapsulamento
  - O Tipo Valor definido pela construção não admite tipos derivados
  - Admite membros de tipo (**`static`**) e de instância (por omissão)
  - Os membros podem definir:
    - Dados: campos
    - Comportamento: métodos, propriedades e eventos
    - Tipos: *Nested Types* (sempre membros estáticos)
    - Acessibilidades (dos membros): **`private`**(por omissão) e **`public`**
  - Acessibilidade (do tipo): **`private`** (por omissão: interno ao *assembly* onde está definido), e **`public`**
  - Semântica de cópia: cópia de todos os campos
  - Herda (de **`System.ValueType`**) uma implementação de **`Equals`** que verifica identidade (e igualdade).

Value Types

In place

Campos de instância

Exemplo de tipo valor: Complex

```
.class public sequential sealed Complex
extends [mscorlib]System.ValueType
```

```
public struct Complex {
    private float real, imaginary;

    public Complex(float real, float imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    public Complex Add(Complex c) {
      return new Complex(real + c.real, imaginary + c.imaginary);
    }


    public override string ToString() {
        return (System.String.Format("{0} + {1}i", real, imaginary));
    }

}
```

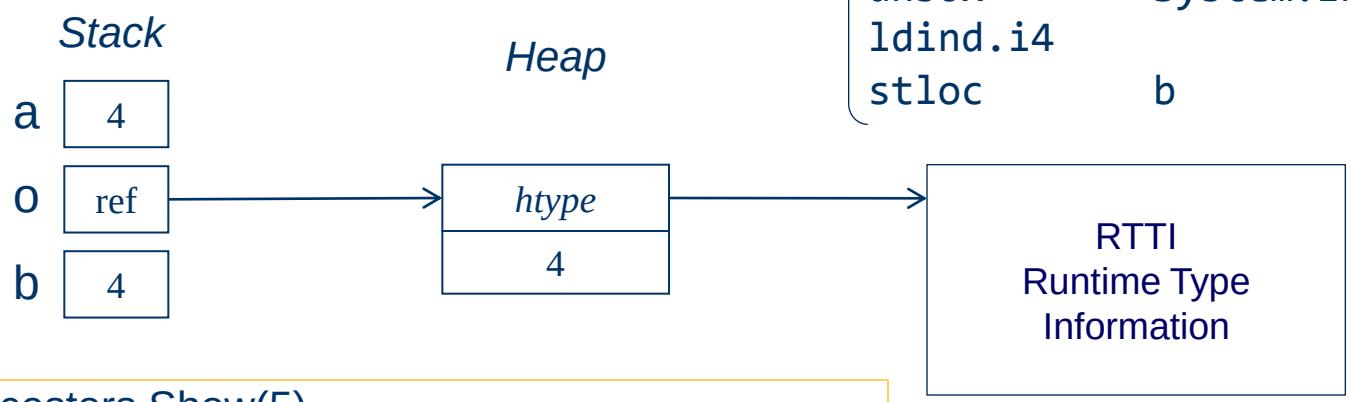
Unificação do sistema de tipos - *Boxing* e *Unboxing*

C#

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

(pseudo) IL

```
ldc.i4.4
stloc       a

ldloc       a
box         System.Int32
stloc       o

ldloc       o
unbox       System.Int32
ldind.i4
stloc       b
```



Ancestors.Show(5)

System.Int32
System.ValueType
System.Object

Linguagem C# - Unificação do sistema de tipos (resumo)

Resumindo:

- As variáveis
  - de Tipos Valor contêm os dados (valores)
  - de Tipos Referência contêm a localização dos dados (valores)
- 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)
  - é sempre possível determinar o seu tipo exacto (*memory safety*)
    - 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*)

Escolher entre *Value Type* e *Reference Type*

- Escolher *Value Type* quando:
  - Não se prevê a necessidade de utilização polimórfica (criação de classes derivadas).
  - Tem poucos dados (valores tipicos de 1 a 16 bytes).Caso tenha mais não é usado regularmente em parâmetros ou retorno de métodos (devido ao custo da passagem de parâmetros por cópia)
  - Usado em cenários de passagem de parâmetros para código *unmanaged*

Demo 1

Tipos Valor vs Tipos Referência

O tipo valor Ponto em C#

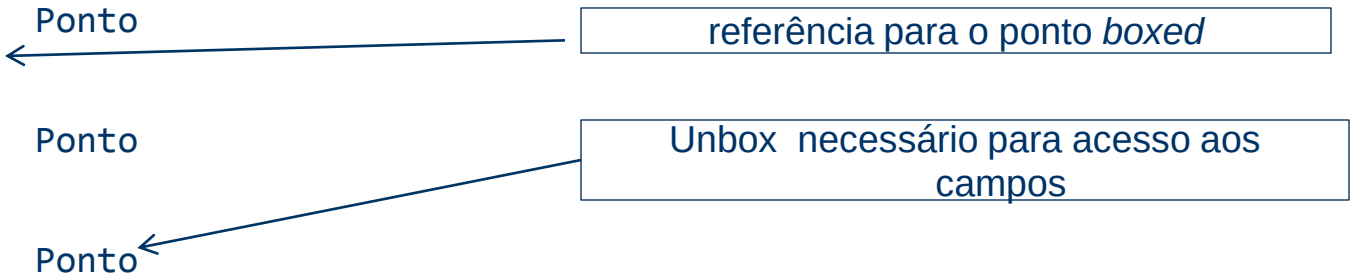
```
public struct Ponto {
  public int x, y;
  public Ponto(int x, int y) { this.x=x; this.y=y; }

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

  public static void Main() {
    Ponto p1 = new Ponto(2,2), p2= new Ponto(2,2);
    object o = p1;
    Ponto p = (Ponto) o;
    p.x=3;
    Console.WriteLine(o.ToString());
    return 0;
  }
}
```

O tipo valor Ponto em C++/CLI

```
public value class Ponto {
public:
  int x, y;

  Ponto(int x, int y) { this->x=x; this->y=y; }

  virtual String^ ToString() override {
      return String::Format("({0},{1})", x, y);
  }

};


int main( array<System::String ^> ^args ) {
    Ponto p1(2,2), p2(2,2);
    Object ^o = p1;
    Ponto ^p = (Ponto ^) o;
    p->x=3;
    Console::WriteLine(o->ToString());
    return 0;
}
```

Excerto do CIL da função main na versão C++/CLI

```
.locals init ([0] object o,
         [1] int32 V_1,
         [2] class Ponto(?) p,
         [3] valuetype Ponto p1,
         [4] valuetype Ponto p2)
IL_0000:  ldc.i4.0
IL_0001:  stloc.1
IL_0002:  ldloca.s   p1
IL_0004:  ldc.i4.2
IL_0005:  ldc.i4.2
IL_0006:  call       instance void Ponto::.ctor(int32,int32)
IL_000b:  ldloca.s   p2
IL_000d:  ldc.i4.2
IL_000e:  ldc.i4.2
IL_000f:  call       instance void Ponto::.ctor(int32, int32)
IL_0014:  ldloc.3
IL_0015:  box        Ponto
IL_001a:  stloc.0
IL_001b:  ldloc.0
IL_001c:  castclass  Ponto
IL_0021:  stloc.2
IL_0022:  ldloc.2
IL_0023:  unbox      Ponto
IL_0028:  ldc.i4.3
IL_0029:  stfld      int32 Ponto::x
IL_002e:  ldloc.0
IL_002f:  callvirt   instance string [mscorlib]System.Object::ToString()
…
```



Boxing na versão 2.0 do framework (prefixo constrained)
standard CLI, secção III-2.1

- The **constrained**. prefix is permitted only on a **callvirt** instruction. The type of *ptr* must be a managed pointer (&) to *thisType.* **The constrained prefix is designed to allow callvirt instructions to be made in a uniform way independent of whether *thisType* is a value type or a reference type.**
- When callvirt *method* instruction has been prefixed by constrained *thisType* the instruction is executed as follows.

  a) If *thisType* is a reference type (as opposed to a value type) then ptr is dereferenced and passed as the 'this' pointer to the callvirt of *method*

  b) If *thisType* is a value type and *thisType* implements *method* then ptr is passed unmodified as the 'this' pointer to a call of *method* implemented by *thisType*

  c) If *thisType* is a value type and *thisType* does not implement *method* then ptr is dereferenced, boxed, and passed as the 'this' pointer to the callvirt of *method*

- This last case can only occur when *method* was defined on System.Object, System.ValueType, or System.Enum and not overridden by *thisType*. In this last case, the boxing causes a copy of the original object to be made, however since all methods on System.Object, System.ValueType, and System.Enum do not modify the state of the object, this fact can not be detected.

Identidade e Igualdade em Value Types – equals de System.ValueType

```
public class ValueType {

   public override bool Equals(object obj) {
        if(obj == null) return false;
        Type thisType = this.GetType();
        if(thisType != obj.GetType()) return false;
        FieldInfo[] fields = thisType.GetFields(BindingFlags.Public
                | BindingFlags.NonPublic | BindingFlags.Instance);
        for(int i=0; i<fields.Length; ++i)
        {
                object thisFieldValue = fields[i].GetValue(this);
                object objFieldValue = fields[i].GetValue(obj);
                if(!object.Equals(thisFieldValue,objFieldValue))
                        return false;
        }
        return true;
   }

}
```

Implementação de Equals em ValueType

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);}

  public static void Main() {
    Ponto p1 = new Ponto(2,2), p2= new Ponto(2,2);
    object o = p1;
    Ponto p = (Ponto) o;
    p.x=3;
    Console.WriteLine(o.ToString());
    return 0;
  }
}
```

Demo com override de Equals em Value Type e Reference Type

- Demo

Sumário de padrões de override de Equals

- Override de `Equals` implica override de `GetHashCode` de forma a manter o invariante:

```
o1.Equals(o2) == true => o1.GetHashCode()== o2.GetHashCode()
```

- A implementação deve verificar se o tipo do objecto passado como parâmetro é igual ao tipo do `this`:

```
public override bool Equals(object obj) {

 if (obj.GetType() != this.GetType())
     return false;
  ....
  return ...;
}
```

Se o tipo for A, o código é equivalente a: `obj is A` ?

- Value Types
  - Override de `Equals` em `ValueType` deve sempre ser feito para evitar a penalização da implementação de `ValueType`
  - Deverá haver a sobrecarga em value type de nome VT:
    - bool Equals(VT v);

- Reference Types
  - Verificar situações de referências nulas
  - Invocar `Equals` da classe base caso esta também tenha `Equals` redefinido

Sobrecarga de operadores

- Algumas linguagens permitem aos tipos definir como certos operadores manipulam as respectivas instâncias.
- Por exemplo: System.String sobrecarrega == e !=.
- Cada linguagem tem os seus próprios operadores e define o seu significado (semântica).
- O CLR não tem qualquer noção de sobrecarga de operadores nem mesmo de operador. Do ponto de vista do CLR a sobrecarga de um operador trata-se apenas de um método estático (com o atributo **specialname**)
- Embora o CLR não tenha noção de operadores, especifica como as linguagens devem expor sua a sobrecarga, por forma a que este mecanismo possa ser usado em diversas linguagens

Nomes recomendados pelo CLR para sobrecarga dos operadores do C# (1)

C# Operator Symbol	*Special Method Name*	*Suggested CLS-Compliant Method Name*
+	op_UnaryPlus	Plus
-	op_UnaryNegation	Negate
~	op_OnesComplement	OnesComplement
++	op_Increment	Increment
--	op_Decrement	Decrement
(none)	op_True	IsTrue { get; }
(none)	op_False	IsFalse {get; }
+	op_Addition	Add
+=	op_AdditionAssignment	Add
-	op_Subtraction	Subtract
-=	op_SubtractAssignment	Subtract
*	op_Multiply	Multiply
*=	op_MultiplyAssignment	Multiply

Nomes recomendados pelo CLR para sobrecarga dos operadores do C# (2)

C# Operator Symbol	*Special Method Name*	*Suggested CLS-Compliant Method Name*
/	op_Divison	Divide
/=	op_DivisonAssignment	Divide
%	op_Modulus	Mod
%=	op_ModulusAssignment	Mod
^	op_ExclusiveOr	Xor
^=	op_ExclusiveOrAssignment	Xor
&	op_BitwiseAnd	BitwiseAnd
&=	op_BitwiseAndAssignment	BitwiseAnd
|	op_BitwiseOr	BitwiseOr
|=	op_BitwiseOrAssignment	BitwiseOr
&&	op_LogicalAnd	And
||	op_LogicalOr	Or
!	op_LogicalNot	Not

Nomes recomendados pelo CLR para sobrecarga dos operadores do C# (3)

C# Operator Symbol	*Special Method Name*	*Suggested CLS-Compliant Method Name*
<<	op_LeftShift	LeftShif
<<=	op_LeftShiftAssignment	LeftShif
>>	op_RightShift	RightShift
>>=	op_RightShiftAssignment	RightShift
(none)	op_UnsignedRightShiftAssignment	RightShift
==	op_Equality	Equals
!=	op_Inequality	Compare
<	op_LessThan	Compare
>	op_GreaterThan	Compare
<=	op_LessThanOrEqual	Compare
>=	op_GreaterThanOrEqual	Compare
=	op_Assign	Assign

O tipo valor Ponto em C# (com overload de operadores)

```
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);}

  public static bool operator ==(Ponto p1, Ponto p2) {
       return Object.Equals(p1, p2);
  }

  public static bool operator !=(Ponto p1, Ponto p2) {
       return !Object.Equals(p1, p2);
  }
}
```

Tipo valor: Complex

```
public struct Complex {
    private float real, imaginary;

    public Complex(float real, float imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    public Complex Add(Complex c) {
      return new Complex(real + c.real, imaginary + c.imaginary);
    }

    // overload do operador +
    public static Complex operator +(Complex c1, Complex c2) {
         return c1.Add(c2);
    }

    public override string ToString() {
        return (System.String.Format("{0} + {1}i", real, imaginary));
    }

}
```

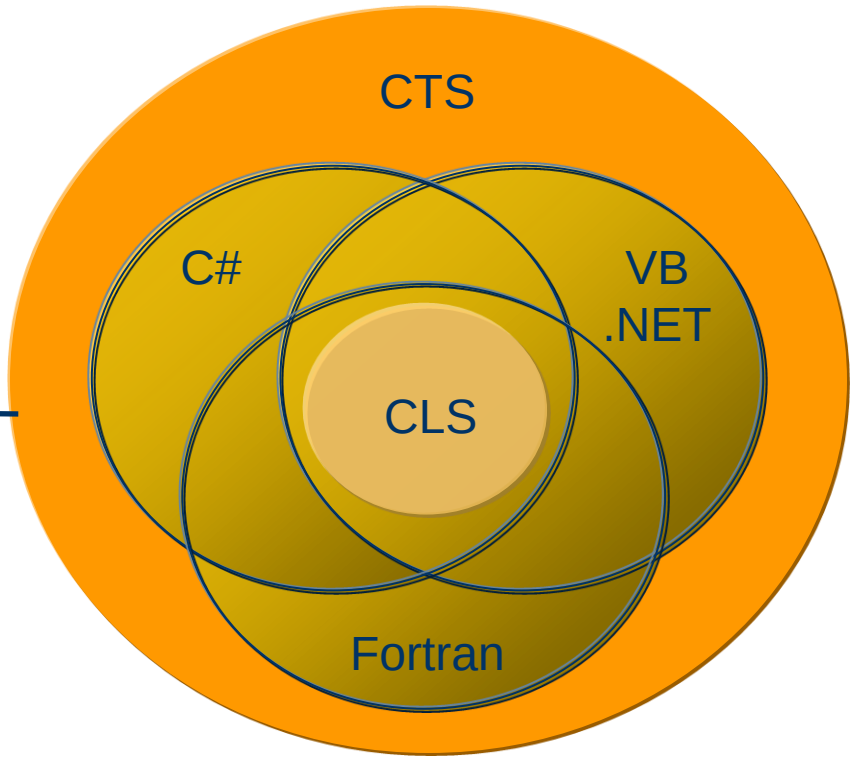
Common Language Specification (CLS)- secção 7 da norma ECMA-335

CLS framework
A library consisting of CLS-compliant code is herein referred to as a *framework. Frameworks are designed for* use by a wide range of programming languages and tools

CLS consumer
A CLS consumer is a language or tool that is designed to allow access to all of the features supplied by CLScompliant
frameworks, but not necessarily be able to produce them.

CLS extender
A CLS extender is a language or tool that is designed to allow programmers to both use and extend CLScompliant frameworks.

Define um conjunto de regras. Os tipos construídos de acordo com essas regras têm a garantia de compatibilidade com qualquer linguagem suportada no .NET



Excerto das regras de criação de tipos CLS

- **Rule 1:** CLS rules apply only to those parts of a type that are accessible or visible outside of the defining assembly (see Section 6.3).
- **Rule 3:** The CLS does not include boxed value types (see clause 7.2.4).
- **Rule 4:** For CLS purposes, two identifiers are the same if their lowercase mappings (as specified by the Unicode locale-insensitive, 1-1 lowercase mappings) are the same.
- **Rule 5:** All names introduced in a CLS-compliant scope shall be distinct independent of kind, except where the names are identical and resolved via overloading. That is, while the CTS allows a single type to use the same name for a method and a field, the CLS does not (see clause 7.5.2).
- **Rule 11:** All types appearing in a signature shall be CLS-compliant (see clause 7.6.1).
- **Rule 12:** The visibility and accessibility of types and members shall be such that types in the signature of any member shall be visible and accessible whenever the member itself is visible and accessible. For example, a public method that is visible outside its assembly shall not have an argument whose type is visible only within the assembly (see clause 7.6.1).
- **CLS Rule 23: System.Object** is CLS-compliant. Any other CLS-compliant class shall inherit from a CLScompliant class.

Construção de Tipos

- Características gerais

Herança

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

Atributos pré-definidos aplicáveis a um tipo

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

- Até à versão 2.0, apenas um dos modificadores pode ser aplicado a um tipo.
  - Para criar um tipo que não suporte derivação e do qual não possam ser criadas instâncias (p.ex. **`System.Console`** ou **`System.Math`**) deve-se:
    - Marcar o tipo como **`Sealed`**
    - Definir somente construtor privado sem parâmetros (em C#)
- Na versão 2.0 é possível marcar um tipo como *abstract* e *sealed* (em C# usa-se a *keyword* `static`). Neste caso não é necessário definir um constructor privado

Visibilidade de um tipo

IL Term	*C# Term*	*Visual Basic Term*	*Description*
`private`	`internal` `(por omissão)`	`Friend`	Visível apenas dentro do *assembly*
`public`	`public`	`Public`	Visível dentro e fora do *assembly*

Tipos *user defined (classes e estruturas)*

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

Acessibilidade dos membros de um tipo

IL Term	*C# Term*	*Visual Basic Term*	*Description*
`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

Tipos aninhados

```
namespace AcmeCorp.LOB {
  public sealed class Customer {
     public sealed class Helper {
       private static int incAmount;
       public static void IncIt() {
          // legal - methods in nested types can access private
          // members of containing type
          nextid += incAmount;
        }
      }

      private static int nextid;
      public static void DoWork() {
         // legal - IncIt is public member
         Helper.IncIt();
         // illegal - incAmount is private
         Helper.incAmount++;
      }
    }
  }
```

Iniciação de tipos

- Construtor de tipo:
  - Tem nome especial → .cctor. O CLR garante a chamada ao construtor de tipo antes de qualquer acesso a um campo de tipo
- Os campos com expressões de iniciação na sua definição, são os primeiros a ser iniciados pelo construtor;
- O método não recebe parâmetros;
- Como sempre, cada linguagem define uma sintaxe para especificar o código deste método.

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)

Demo

- Iniciação de Tipos

Instâncias

- Instâncias

Construção de objectos

- O CLR requer que todos os objectos sejam criados usando o operador *new* (que emite a instrução IL *newobj*). A seguinte linha mostra como se cria um objecto *Employee*:

```
Employee e = new Employee("ConstructorParam1");
```

- O operador *new* faz o seguinte:
  1. Reserva, no *managed heap*, um bloco de memória com o número de *bytes* necessários para armazenar o objecto do tipo especificado.
  2. 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, e o segundo é o *SyncBlockIndex* (usado na sincronização).
  3. É chamado o construtor de instância do tipo, passando os argumentos (o *string* «*ConstructorParam1*» no exemplo) especificados na invocação de *new*. (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.)
- Depois de *new* ter realizado estas operações, devolve uma referência para o objecto que acabou de criar. No exemplo de código, esta referência é salva na variável **`e`**, cujo tipo é *Employee*.

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.

- Métodos virtuais no construtor:
  - As chamadas a métodos virtuais nos construtores seguem as mesmas regras que em qualquer outra circunstância.

Em geral, deverão ser evitadas as chamadas a métodos virtuais em construtores, pois o `this`, por definição, ainda não está completamente construído

Construtores de Value Types em C#

- Os *value types* não têm construtor por omissão
- Os *value types* não permitem construtores sem parâmetros

Métodos

- Métodos

Atributos pré definidos aplicáveis a métodos

CLR Term	*C# Term*	*Visual Basic Term*	*Description*
`Static`	`static`	`Shared`	*Method is associated with the type itself, not an instance of the type. Static members can't access instance fields or methods defined within the type because the static method isn't aware of any object.*
`Instance`	`(default)`	`(default)`	*Method is associated with an instance of the type, not the type itself. The method can access instance fields and methods as well as static field and methods.*
`hidebysig`			*esconde o método com igual assinatura - política do C#.* *A omissão do atributo faz com que o método esconda todos as sobrecargas de métodos com o mesmo nome (hide-by-name) - (política do C++)*
`Virtual`	`virtual`	`Overridable`	*Most-derived method is called even if object is cast to a base type. Applies only to instance methods..*

Atributos pré definidos aplicáveis a métodos virtuais

CLR Term	*C# Term*	*Visual Basic Term*	*Description*
NewSlot	**new (default)**	**Shadows**	*Method should not override a virtual method defined by its base type; the method hides the inherited method.* `NewSlot` *applies only to virtual methods*
	override	**Overrides**	*Explicitly indicates that the method is overriding a virtual method defined by its base type. Applies only to virtual methods.*
Abstract	**abstract**	**MustOverride**	*Indicates that a deriving type must implement a method with a signature matching this abstract method. A type with an abstract method is an abstract type. Applies only to virtual methods.*
Final	**sealed**	**NotOverridable**	*A derived type can't override this method. Applies only to virtual methods.*

Métodos virtuais

- Métodos virtuais podem ser redefinidos ou escondidos nas classes derivadas

```
class Phone {
    public void Dial() { EstablishConnection(); }
    protected virtual void EstablishConnection() { /*...*/ }
}
class AnotherPhone: Phone {
    protected override void EstablishConnection() { /*...*/ }
}

class YetAnotherPhone : Phone {
    public new void Dial() {
        // estabelece outro tipo de ligação
    }

    protected new virtual void EstablishConnection() {
        // Esconde implementação de Phone.EstablishConnection()
    }
}
```

Passagem de parâmetros e retorno de resultados de métodos

- Passagem por valor:
  - Através da cópia do conteúdo da variável;
  - Comportamento por omissão.
- Passagem por referência:
  - Através de ponteiro *managed* para a variável;
  - Em IL indicado por **`&`** e em C# por **`ref`**.
- Retorno de resultado:
  - Por retorno da função;
  - Por parâmetro com **`[out]`** em IL e **`out`** em C#.

Não faz parte da assinatura do método

Parâmetros ref,out e a compatibilidade de tipos

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

Seja duas referência para string:
Qual o problema do seguinte código?

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

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 (e do C#) permitem sobrecarga se os métodos diferirem apenas em número ou tipo dos parâmetros.

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!");
   }
}
```

Campos e Propriedades

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 e campos readonly – C#

- **Constantes**
  - 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*** – literal significa que 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.).
  - Em C# **const Int32 SomeConstant = 1;**
- **Campos readonly**
  - Só podem ser afectados duranta a construção da instância do tipo onde estão definidos
  - Em C#: **readonly Int32 SomeReadOnlyField = 2;**
  - O campo é marcado com o atributo *InitOnly*

Popriedades sem e com parâmetros (indexers) em C#

Property	Indexer
Identified by its name.	Identified by its signature.
Accessed through a simple name or a member access.	Accessed through an element access.
Can be a static or an instance member.	Must be an instance member.
A **get** accessor of a property has no parameters.	A **get** accessor of an indexer has the same formal parameter list as the indexer.
A **set** accessor of a property contains the implicit **value** parameter.	A **set** accessor of an indexer has the same formal parameter list as the indexer, in addition to the **value** parameter.

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
```

Indexers (criação)

```
// Class to provide access to a large file as if it were a byte array.
public class FileByteArray {
    Stream stream;      // Holds the underlying stream

    public FileByteArray(string fileName)  {
        stream = new FileStream(fileName, FileMode.Open);
    }

    // Close the stream. This should be the last thing done when you are finished.
    public void Close()  {
        stream.Close();vstream = null;
    }

    public byte this[long index]  { // long is a 64 bit integer
        get  {
            byte[] buffer = new byte[1];
            stream.Seek(index, SeekOrigin.Begin);
            stream.Read(buffer, 0, 1);
            return buffer[0];
        }
        // Write one byte at offset index and return it.
        set  {
            byte[ ] buffer = new byte[1] {value};
            stream.Seek(index, SeekOrigin.Begin);
            stream.Write(buffer, 0, 1);
        }
    }

    public long Length   {
        get { return stream.Seek(0, SeekOrigin.End); }
    }
}
```

Indexers (Utilização)

```
// Demonstrate the FileByteArray class.  Reverses the bytes in a file.
public class Reverse  {
  public static void Main(String[] args)  {
    // Check for arguments.
    if (args.Length == 0)      {
      Console.WriteLine("indexer <filename>");
      return;
    }

    FileByteArray file =
       new FileByteArray(args[0]);
    long len = file.Length;

    // Swap bytes in the file to reverse it.
    for (long i = 0; i < len / 2; ++i)  {
      byte t;
      // Note that indexing the "file" variable invokes the
      // indexer on the FileByteStream class, which reads
      // and writes the bytes in the file.
      t = file[i];
      file[i] = file[len - i - 1];
      file[len - i - 1] = t;
    }
    file.Close();
  }
}
```

Ficheiro Inicial

```
public class Hello1 {
   public static void Main() {
      System.Console.
      WriteLine("Hello, World!");
   }
}
```

Ficheiro Final

```
}
}
;)"!dlroW ,olleH"(eniLetirW.elosnoC.metsyS
{
)(niaM diov citats cilbup
{
1olleH ssalc cilbup
```

Classes parciais

- 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*

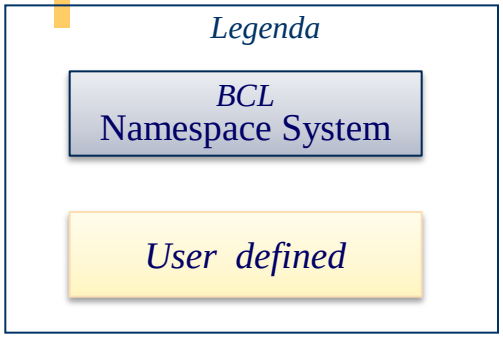
Classes parciais

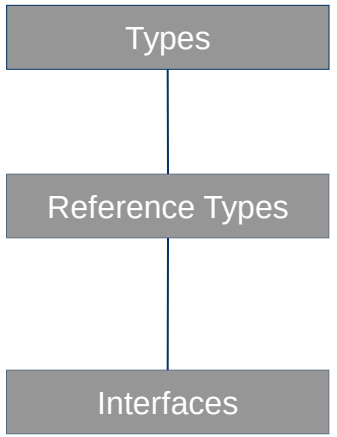
- Aspectos acumulativos de uma classe:
  - Campos
  - Métodos
  - Propriedades
  - Indexadores
  - Interfaces implementadas
- Aspectos não acumulativos:
  - Classe base
  - Tipo-valor ou tipo-referência
  - Visibilidade
- As diversas partes de uma mesma classe devem concordar nos aspectos não acumulativos.

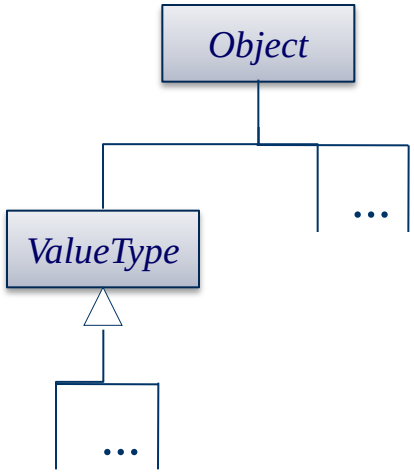
Interfaces

- 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`.
- Em C# a implementação de uma interface resulta, por omissão, em métodos *sealed*.
- Por convenção, o nome das interfaces começa pelo carácter '`I`'
  - Exemplos: **`ICloneable`**, **`IEnumerable`**

Linguagem C# - Excerto do modelo de tipos (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);
```

Implementação explícita de Interfaces (I)

Os tipos que as implementam podem dar implementação privada a alguns dos métodos da interface, útil, por exemplo, em cenários de implementação de duas interfaces que partilham métodos com a mesma assinatura

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

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

Implementação explicita 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 visivel com uma referência para ICowboy
    void ICowboy.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:

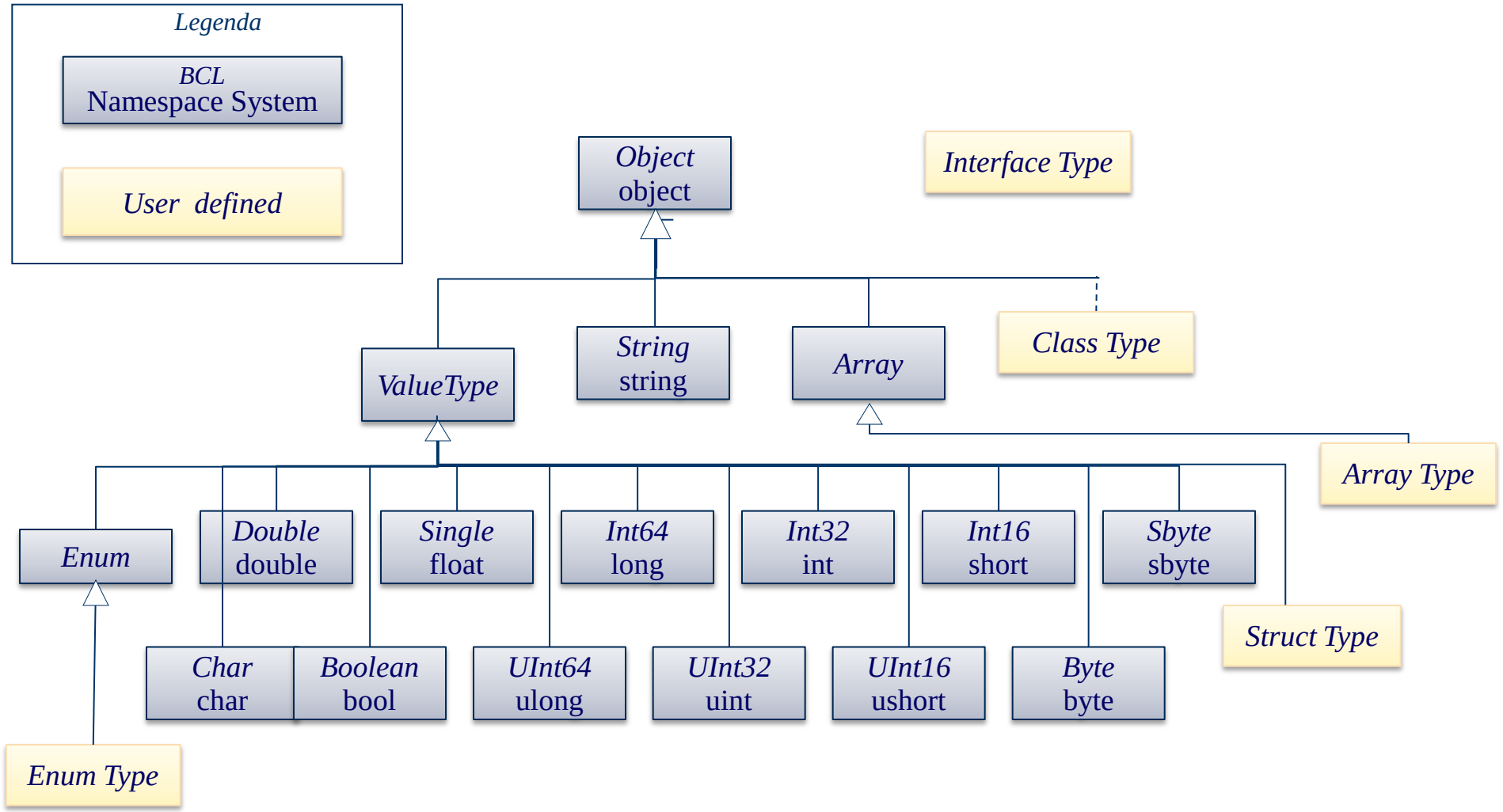
```
Val v = new Val(5);
Val v1 = v.Clone();
```

Sem operações de *box* e *unbox*

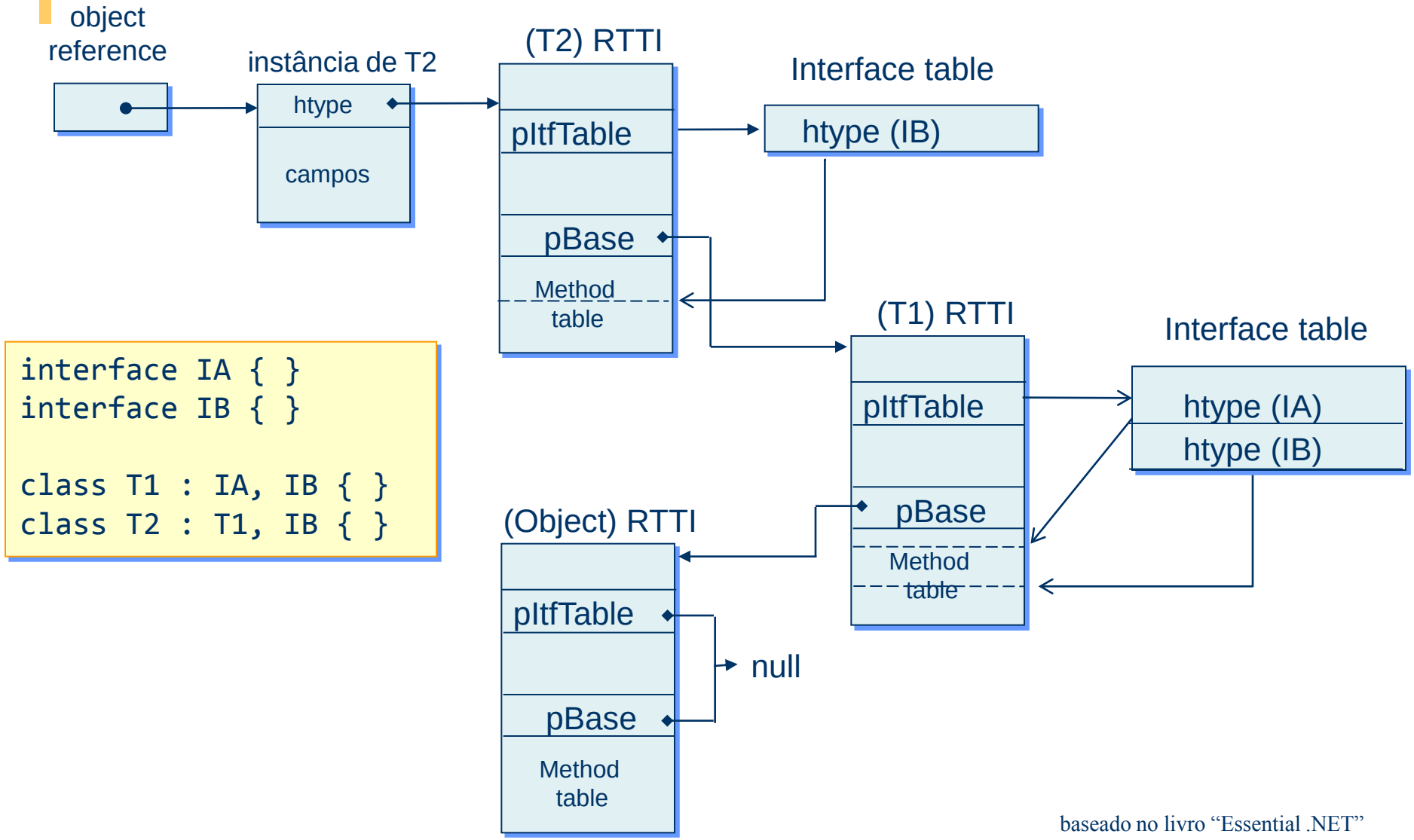
Demo 2

Classes, interfaces, herança, polimorfismo

Linguagem C# - Modelo de tipos revisitado



Informação de tipo em tempo de execução (RTTI)



baseado no livro "Essential .NET"