Ambientes Virtuais de Execução

Common Type System (CTS)

Common Type System (CTS)

- Sistema de tipos (orientado aos objectos) que permite representar tipos de várias linguagens.
  - Idealmente deveria constituir a UNIÃO dos sistemas de tipos das linguagens a suportar.

- 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 genéricos

Tipos Valor (Value Types)

- Os tipos valor contêm directamente os seus dados, e as suas instâncias estão ou alocadas na *stack* ou alocadas *inline* numa estrutura.
- São passados por valor.
- Incluem primitivas, estruturas e enumerações. Exemplos:

```
int i; //primitiva
struct Point { int x, y;} //estrutura
enum State {Off, On} //enumeração
```

- Podem ser definidos novos tipos valor, definindo uma nova classe como derivada da classe `System.ValueType`.
- Os tipos valor são *sealed*, isto é, não podem ser tipos bases para outro tipo valor ou tipo referência.

Tipos Refêrencia (Reference Types)

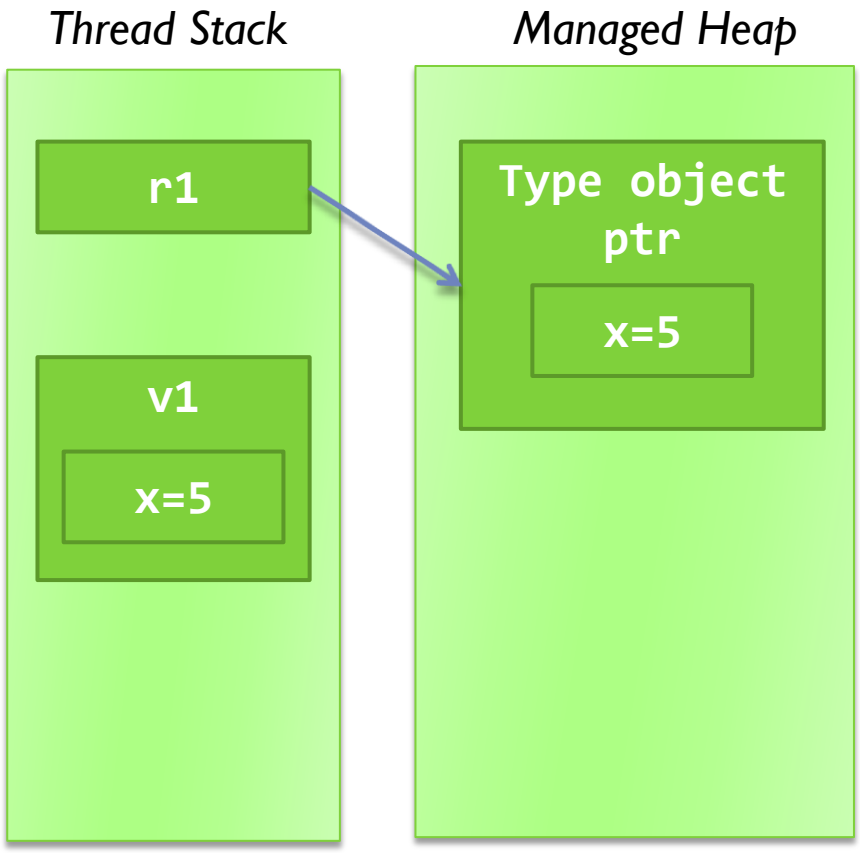
- Os tipos refêrencia representam referências para objectos armazenados no *heap*.
- São passados por referência.
- Os tipos referência incluem classes, interfaces, *arrays* e *delegates*. Exemplos:

```
class Car {} // Class
interface ISteering {} // Interface
int[] a = new int[5]; // Array
delegate void Process( ); // Delegate
```

Tipos Valor versus Tipos Referência - Memória

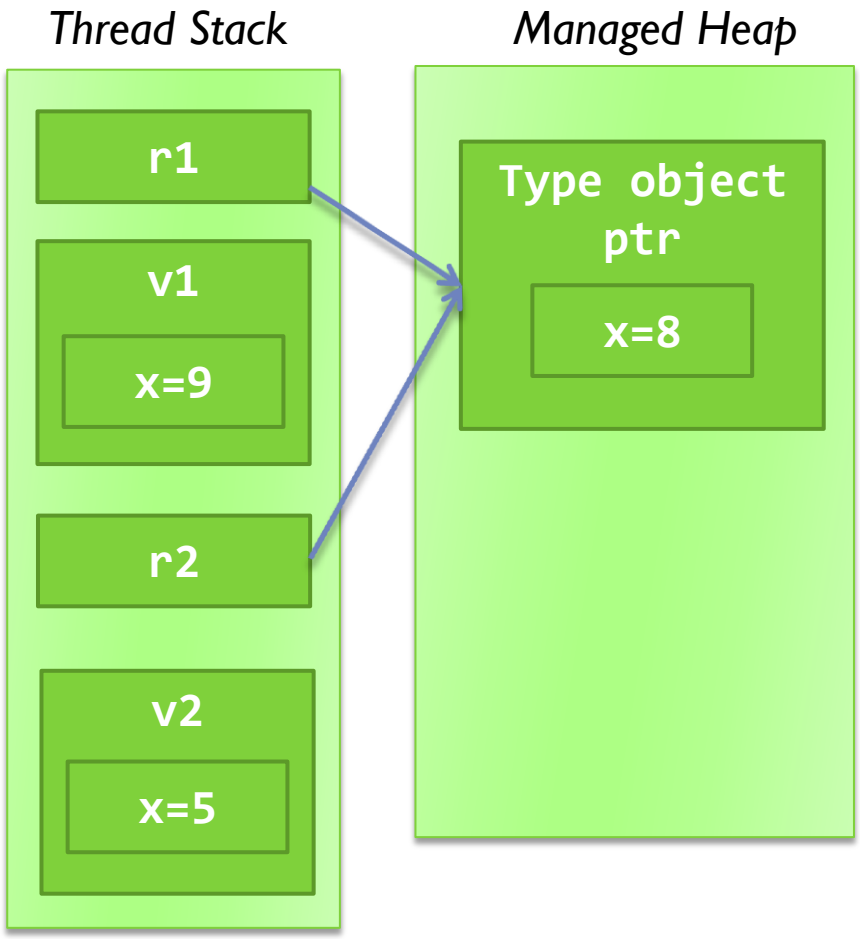
```
//Reference Type
class SomeRef { public Int32 x; }

//Value Type
struct SomeVal{ public Int32 x; }
static void ValueTypeDemo( ){
 SomeRef r1=new SomeRef( );
 SomeVal v1=new SomeVal( );
 r1.x=5;
 v1.x=5;
 Console.WriteLine(r1.x);
 Console.WriteLine(v1.x);
//...
```

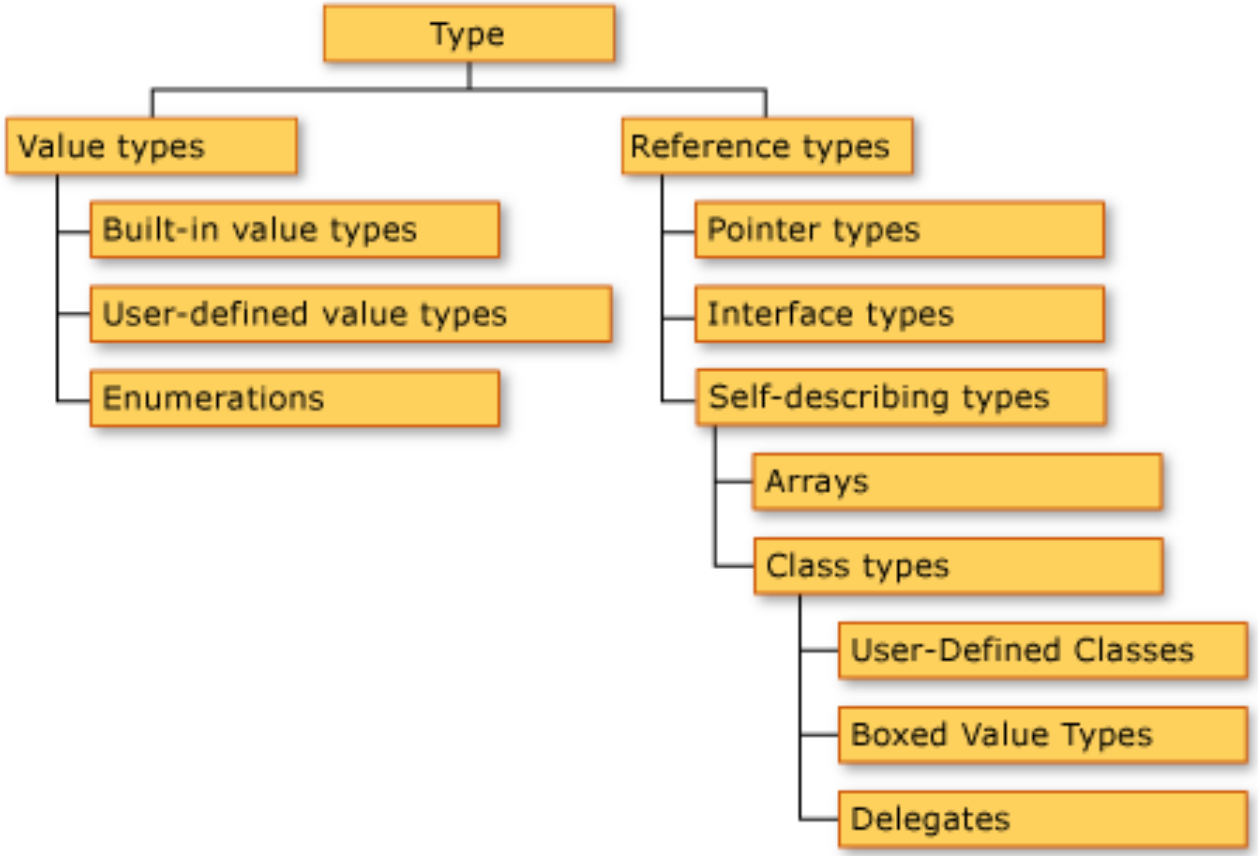


Tipos Valor versus Tipos Referência - Memória

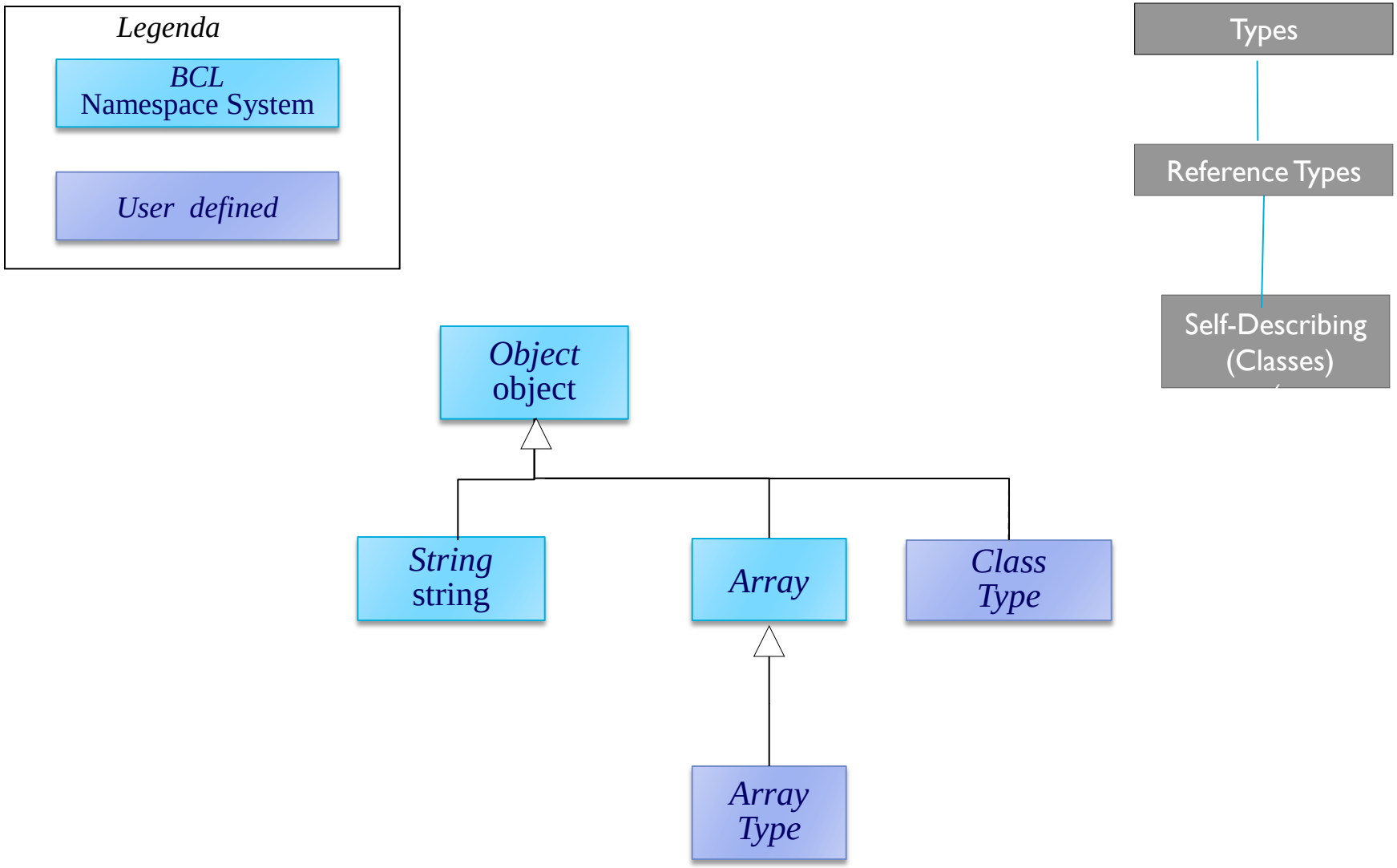
```
//...
SomeRef r2 = r1;
SomeVal v2 = v1;
r1.x = 8;
v1.x = 9;
Console.WriteLine(r1.x);
Console.WriteLine(r2.x);
Console.WriteLine(v1.x);
Console.WriteLine(v2.x);
//...
```



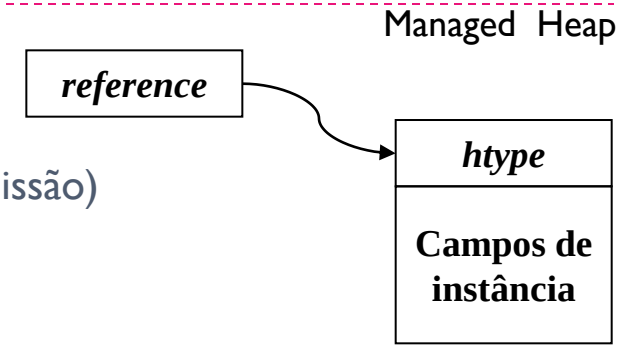
Categoria de Tipos



Excerto do modelo de tipos - classes



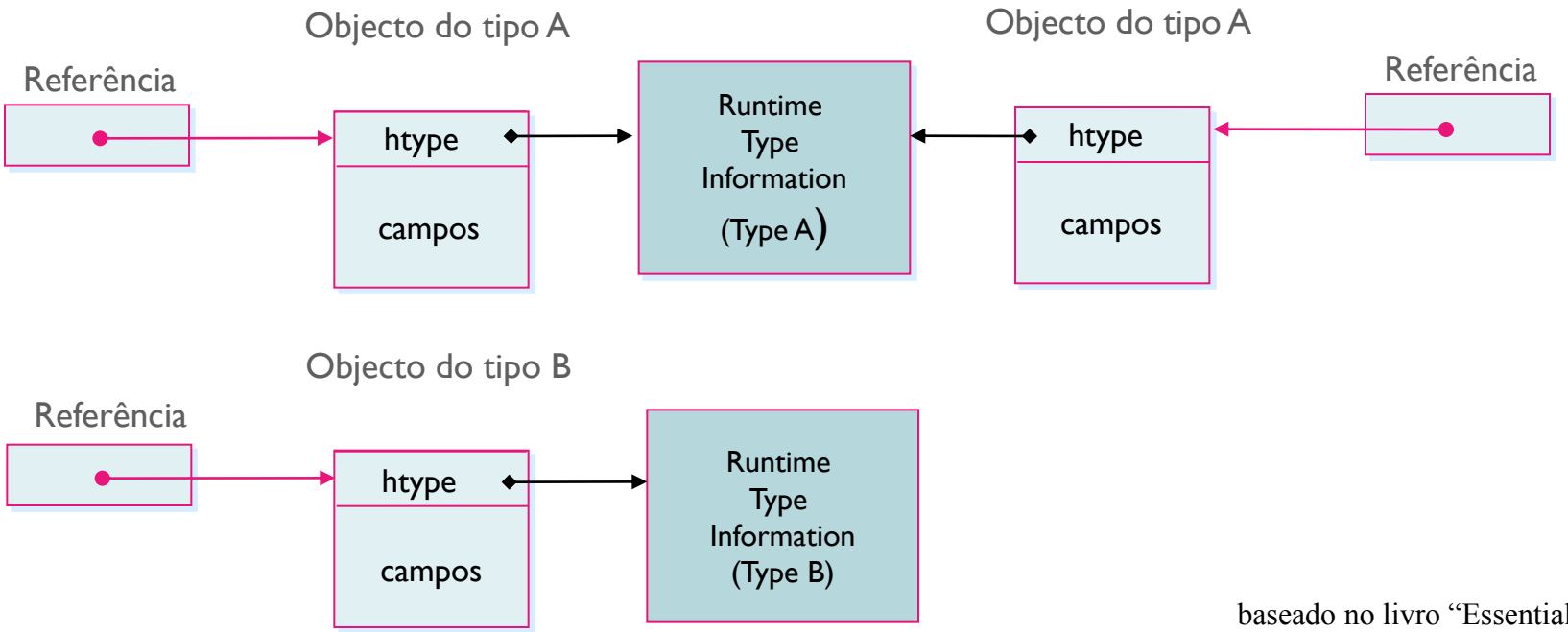
Linguagem C# - construção *de tipos (classes)*



- 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`**), 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

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"

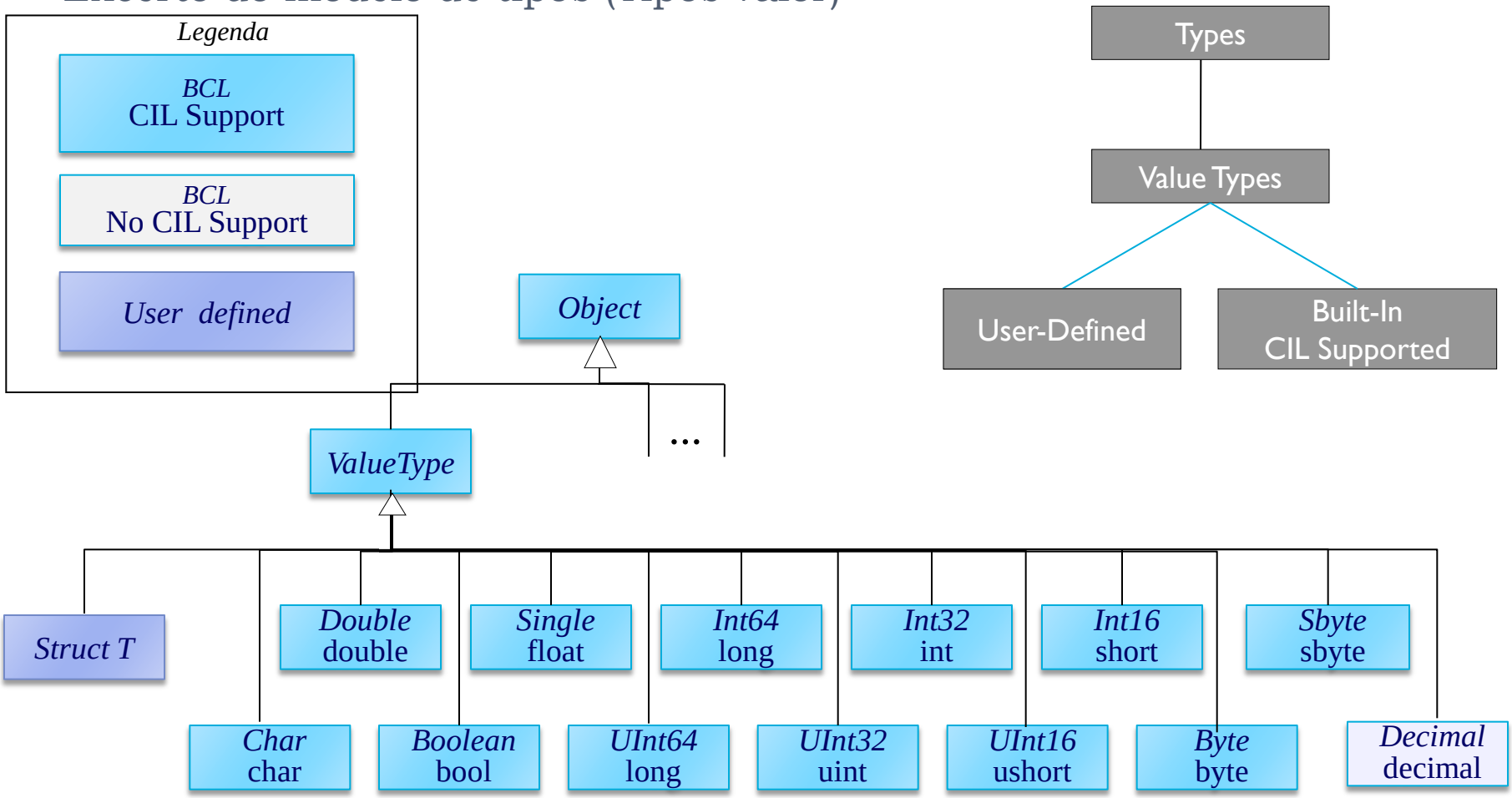
System.Object

- Todos os tipos derivam de `System.Object`.
  - A plataforma .NET suporta tipos valor por razões de performance.
  - Todos os tipos primitivos têm classes correspondentes na plataforma .Net. Por exemplo:
    - A classe correspondente a `int` é `System.Int32`
    - `System.Int32` deriva de `System.ValueType`,
    - Tipos valor podem ser sempre convertidos para um *reference type* – mecanismo designado por Boxing.
    - Unificação do sistema de tipos.

Primitivas C# com os tipos FCL correspondentes

FCL	C#	Descrição	CLS-Compliant	Ref / Value
System.Object	object	Tipo base de todos os tipos	Sim	Ref
System.String	string	Um array de caracteres	Sim	Ref
System.Boolean	bool	True/false	Sim	Val
System.Char	char	Unicode 16-bit char	Sim	Val
System.Single	float	IEC 60559:1989 32-bit float	Sim	Val
System.Double	double	IEC 60559:1989 64-bit float	Sim	Val
System.SByte	sbyte	Signed 8-bit integer	Sim	Val
System.Byte	byte	Unsigned 8-bit integer	Não	Val
System.Int16	short	Signed 16-bit integer	Sim	Val
System.UInt16	ushort	Unsigned 16-bit integer	Não	Val
System.Int32	int	Signed 32-bit integer	Sim	Val
System.UInt32	uint	Unsigned 32-bit integer	Não	Val
System.Int64	long	Signed 64-bit integer	Sim	Val
System.UInt64	ulong	Unsigned 64-bit integer	Não	Val
System.Decimal	decimal	Valores decimais com precisão estendida	Sim	Val

Excerto do modelo de tipos (Tipos valor)



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

- Keyword `struct` (para definição de Tipos Valor)

Value Types

In place

Campos de instância

  - 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).

Conversão entre tipos

- Em tempo de execução CLR sabe sempre, através do método `GetType`, de que tipo um objecto é.
  - É um método que não é virtual.
- CLR permite a conversão de um objecto para o seu tipo ou para um dos seus tipos base (Conversão Implícita).
  - São conversões seguras.
- CLR permite a conversão de um objecto para um dos seus tipos derivados (Conversão Explicita)
  - A conversão pode falhar em tempo de execução.

Conversão entre tipos - Exemplo

```
using System;
class ClassA { public int i = 0; }
class ClassB { public double d = 0; }
class Program{
  public static void Main( ){
     Object o = new ClassA( );
     ClassB b = new ClassB( );
     xpto( b );
  }
  public static void Xpto(Object o){
   ClassA a = ( ClassA ) o;
  }
}
```

Conversão com C#
Operadores `is` e `as`

- O operador `is` verifica se um objecto é compatível com um dado tipo.
  - Exemplo:

```
Object o;
//...
if (o is ClassA){
    ClassA a = (ClassA) o;
}
//...
```

- O operador `as` verifica se um objecto é compatível com um dado tipo. Se for, faz a conversão, caso contrário retorna `null`.
  - Só se verifica o tipo do objecto uma vez.
  - Exemplo:

```
Object o;
//...
ClassA a = o as ClassA;
//...
```

Conversões implícitas e explícitas entre tipos

```
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)

- 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#:
    - Coersão com alargamento (widening): implícita
    - Coersão com diminuição de resolução (narrowing): explícita
    - Exemplos:

```
Int32 i = 5;
Int64 l = 5;
Single s = i;
Byte b = (Byte) i;
Int16 v = (Int16) s;
```

Operações checked e unchecked

- CLR tem instruções IL que permitem o compilador realizar ou não verificação de *overflow*
  - `add, sub, mul, conv` realizam operações sem verificação;
  - `add.ovf, sub.ovf, mul.ovf, conv.ovf` verficam e caso ocorra *overflow* lançam excepção do tipo `System.OverflowException.`
- Por omissão, em C# não é feita verificação de *overflow*.
- Em C# o operador *checked* garante a verificação de *overflow*.

  Exemplo:

```
//...
  int i =1024
  byte b = 0
  checked{ b=(byte) i;}
//...
```

System.OverflowException: Arithmetic operation resulted in an overflow

- O operador *unchecked* garante a não verificação de *overflow*.

Conversão entre tipos numéricos (coerção)

```
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
```

boxing e unboxing – Exemplo 1

C#

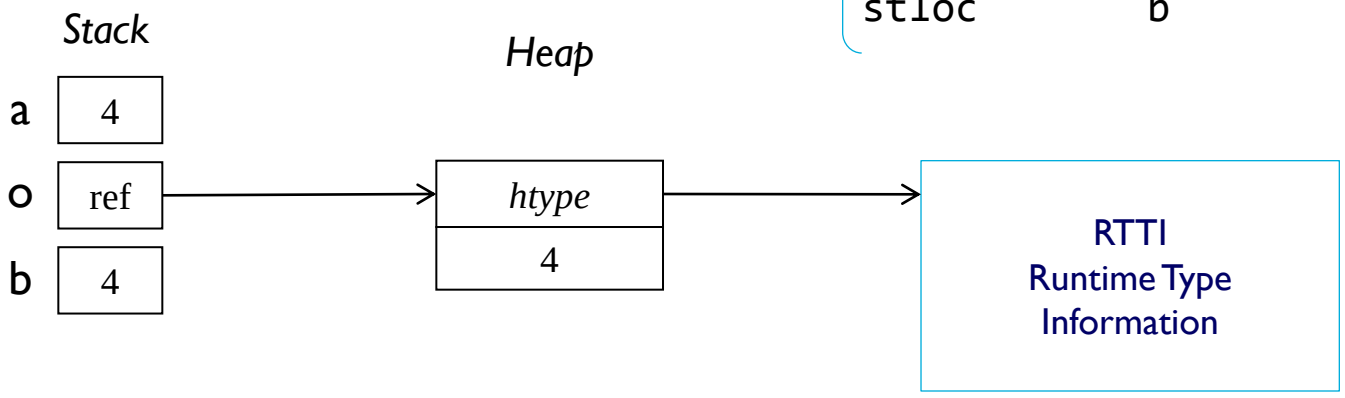
(pseudo) IL

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

```
ldc.i4.4
stloc       a

ldloc       a
box         System.Int32
stloc       o

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



Boxing e unboxing – Exemplo 3

C#

```
public sealed class Program{
  public static void main(){
    Int32 x = 5;
    Object o = x;
    Int16 z = (Int16)(Int32) o;
    Int16 y = (Int16) o; //InvalidCastException

}
```

O *unboxing* de um objecto obriga a que a conversão se faça para o tipo valor exacto.

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*
  - Não se prevê utilização em colecções.

System.Object

```
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

bool 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
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*.
string 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.
Type GetType()	Retorna um objecto de um tipo derivado de `Type` que identifica o tipo do objecto invocado
object MemberwiseClone()	Retorna um *clone* do objecto invocado por *shallow copy* (cópia não recursiva)
void Finalize()	Usado no processo de recolha automática de memória

O tipo valor Ponto em C# (com redefinição do ToString)

```
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());
  }
}
```

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;
}
```

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.

Unificação do sistema de tipos - *Exemplo*

```
class Ancestors {

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

```
Ancestors.Show(5.GetType())

  System.Int32
  System.ValueType
  System.Object
```

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 V2 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).

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;
  }
}
```

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 ...;
}
```

- 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

O tipo valor Ponto em C# (continuação)

```
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);
         Console.WriteLine(p1.ToString());
   //..
   } // ...
}
```

Excerto do método toString em IL

```
//...
    IL_0017:  box      [mscorlib]System.Int32
    IL_001c:  call     string [mscorlib]System.String::Format(string, object, object)
//..
```

Excerto do método main em IL

```
//...
IL_000d:  constrained. Ponto
IL_0013:  callvirt   instance string [mscorlib]System.Object::ToString()
IL_0018:  call       void [mscorlib]System.Console::WriteLine(string)
//...
```

O tipo valor Ponto em C# (continuação)

```
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);
         Console.WriteLine(p1);
   //..
   } // ...
}
```

Excerto do método toString em IL

```
//...
    IL_0017:  box      [mscorlib]System.Int32
    IL_001c:  call     string [mscorlib]System.String::Format(string, object, object)
//..
```

Excerto do método main em IL

```
//...
 IL_000c:  box        Ponto
 IL_0011:  call       void [mscorlib]System.Console::WriteLine(object)
//...
```

O tipo valor Ponto em C# (continuação)

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

   public static void Main( ){
      Ponto p = new Ponto(2,2);
      Console.WriteLine(p.GetType());
  // ...
  }
 //...
}
```

Excerto do método main em IL

```
//...
IL_000c:  box  Ponto
IL_0011:  call instance class [mscorlib]System.Type[mscorlib]System.Object::GetType()
IL_0016:  call       void [mscorlib]System.Console::WriteLine(object)
//..
```

Métodos herdados e Tipos Valor

- Os métodos virtuais herdados ou redefinidos pelo tipo valor (Equals, GetHashCode, ToString) podem ser invocados pelos mesmos:
  - CLR pode invocar estes métodos não virtualmente
  - System.ValueType redefine todos estes métodos virtuais
    - É esperado que o valor do argumento this se refira a uma instância de tipo valor unboxed.
- A invocação de métodos herdados não virtuais requere o boxing do tipo valor
  - Estes métodos esperas que o argumento this se refira a um objecto no heap.

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 oveload 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));
    }

}
```

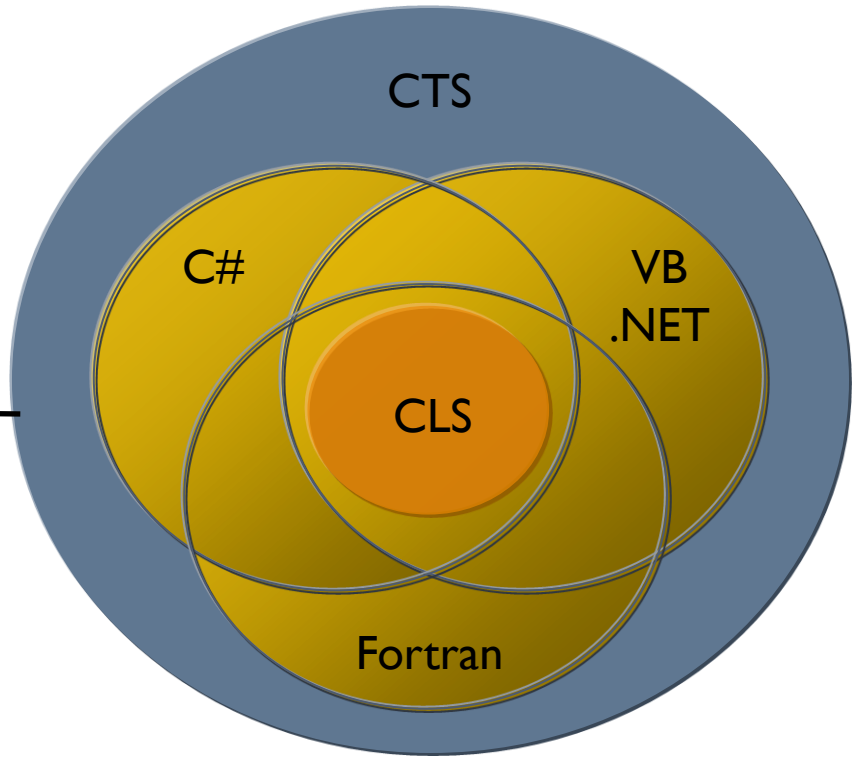
Common Language Specification (CLS)- secção 7 da norma ECMA-335

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.

Namespaces e *Assemblies*

- Os *namespaces* permitem o agrupamento lógico de tipos relacionados. Este mecanismo é usado pelo programadores para localizarem facilmente um determinado tipo. Por exemplo, o *namespace System.Collections* define o grupo de tipos colecção, e o *namespace System.IO* define o grupo de tipos relacionados com as operações de I/O. A seguir apresenta-se código que constrói um objecto do tipo *System.IO.FileStream* e um objecto do tipo *System.Collections.Queue*:

```
class App {
  static void Main() {
    System.IO.FileStream fs = new System.IO.FileStream(...);
    System.Collections.Queue q = new System.Collections.Queue();
  }
}
// ou
using System.IO;              // Try prepending "System.IO"
using System.Collections;     // Try prepending "System.Collections"
class App {
  static void Main() {
    FileStream fs = new FileStream(...);
    Queue q = new Queue();
  }
}
```

Importante O CLR não tem a noção de *namespaces*. Quando acede a um tipo, o CLR necessita de conhecer o nome completo do tipo e qual o assembly que contém a respectiva definição.
Não existe nenhuma relação entre *assemblies* e *namespaces*.