

Metadata

- ◆ Na Programação por Componentes, o componente necessita ser auto-descritivo nas várias fases da sua utilização:
 - Em **tempo de desenho** (Ex: utilização em ambientes RAD)
 - Em **tempo de carregamento** para ser escolhido o ambiente de execução adequado (Ex: execução *thread safe* de métodos, etc.)
 - Em **tempo de execução** (Ex: suporte a serialização)
- ◆ A auto-descrição é feita na forma de metadata obrigatoriamente associada ao componente

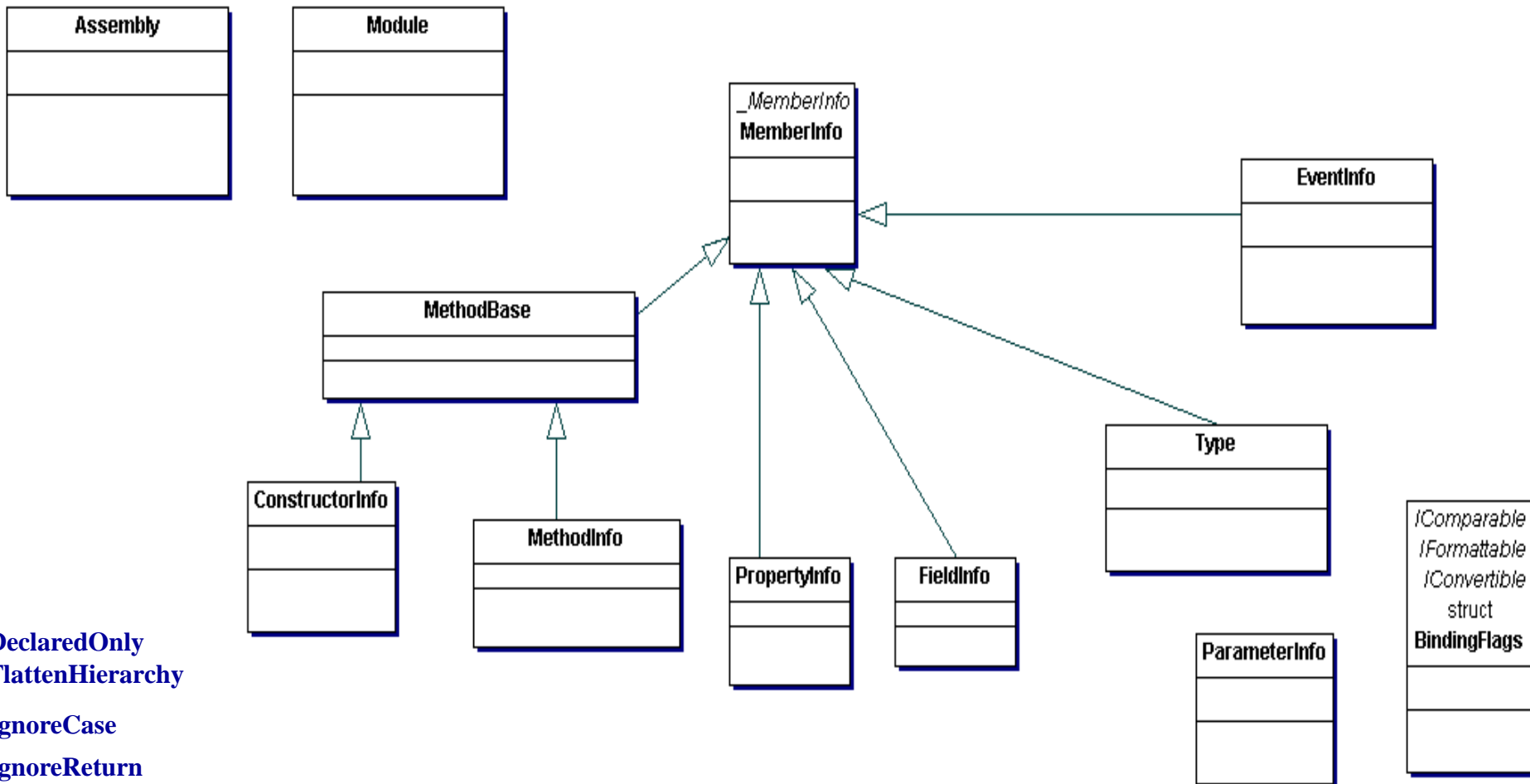
◆ **System.Reflection**

- Inspeccionar o Sistema de Tipos
- Criar instâncias de tipos apenas conhecidos em tempo de execução. Alterar os seus campos e propriedades. Invocar métodos de instância e de tipo.

◆ **System.Reflection.Emit**

- Estender o Sistema de Tipos (gerar dinamicamente *assemblies*, criando novos tipos em tempo de execução)

Namespace Reflection (excerto)



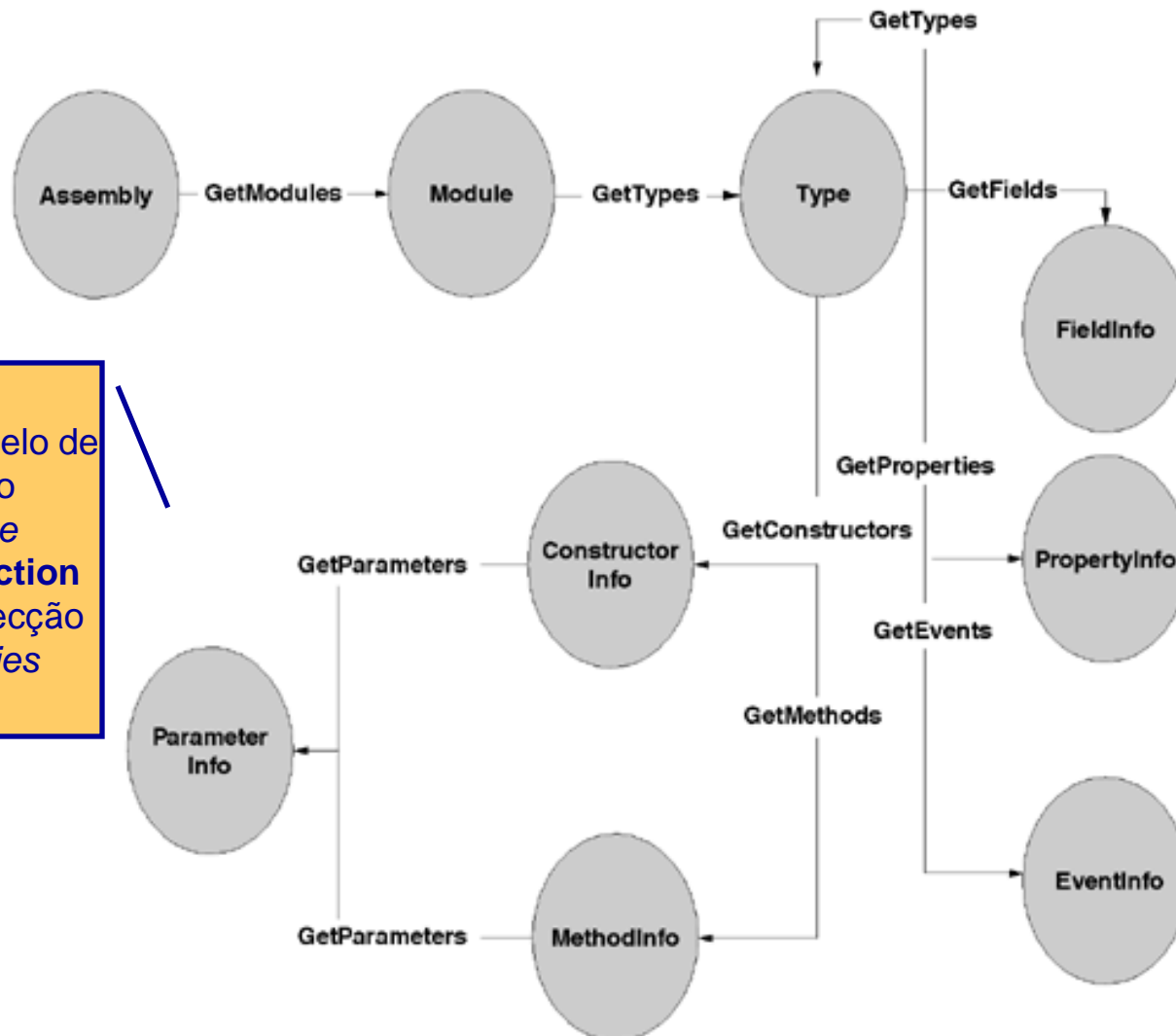
DeclaredOnly
FlattenHierarchy

IgnoreCase
IgnoreReturn

Instance
NonPublic

Public
Static

Reflection



Excerto do modelo de
objectos do
namespace
System.Reflection
usado na inspecção
de *assemblies*

class Assembly

```
class Assembly {
    public Type[] GetExportedTypes();
    public Module GetModule(String name);
    public AssemblyName[] GetReferencedAssemblies();
    public AssemblyName GetName();
    public AssemblyName GetName(Boolean copiedName);
    public static Assembly GetAssembly(Type type);
    public Type GetType(String name);
    public Type GetType(String name, Boolean throwOnError);
    public Type[] GetTypes();
    public static Assembly Load(String assemblyString);
    public static Assembly LoadFile(String path);
    public Object CreateInstance(String typeName);
    public Module[] GetLoadedModules();
    public Module[] GetModules();
    public Module[] GetModules(Boolean getResourceModules);
    public static Assembly GetExecutingAssembly();
    public static Assembly GetCallingAssembly();
    public static Assembly GetEntryAssembly();

    String CodeBase{get;}
    String FullName{get;}
    MethodInfo EntryPoint{get;}
    Module ManifestModule{get;}
}
```

class Module

```
class Module {  
    public System.Type  
        GetType(System.String className, System.Boolean ignoreCase);  
    public System.Type GetType(System.String className);  
    public System.Type[] GetTypes();  
  
    System.String Name{get;}  
    System.Reflection.Assembly Assembly{get;}  
}
```

Type

◆ Type é o tipo fundamental da introspecção:

- A instância de Type que representa o tipo de um objecto ou valor pode ser obtido invocando o método GetType do objecto ou valor.
- Dado um nome de tipo, a instância (única) de Type que o representa é obtida em C# via o operador **typeof** : *typeof (<TypeName>)*.
- A partir de uma instância de Type é possível reconstituir toda hierarquia de tipos a que pertence o tipo representado por aquela instância

```
using Reflection;

public static void ListAllMembersInEntryAssembly() {
    Assembly assembly = Assembly.GetEntryAssembly();
    foreach (Module module in assembly.GetModules())
        foreach (Type type in module.GetTypes())
            foreach (MemberInfo member in type.GetMembers(flags))
                Console.WriteLine("{0}.{1}", type, member.Name );
}
```

Excerto da classe Type (métodos)

```
class Type : MemberInfo {
    public static Type GetType(String typeName);
    public Boolean IsSubclassOf(Type c);
    public Boolean IsAssignableFrom(Type c);
    public ConstructorInfo GetConstructor(Type[] types);
    public ConstructorInfo[] GetConstructors();
    public ConstructorInfo[] GetConstructors(BindingFlags bindingAttr);
    public MethodInfo GetMethod(String name, BindingFlags bindingAttr);
    public MethodInfo GetMethod(String name);
    public MethodInfo[] GetMethods();
    public MethodInfo[] GetMethods(BindingFlags bindingAttr);
    public FieldInfo GetField(String name, BindingFlags bindingAttr);
    public FieldInfo[] GetFields();
    public FieldInfo[] GetFields(BindingFlags bindingAttr);
    public Type GetInterface(String name);
    public Type[] GetInterfaces();
    public EventInfo GetEvent(String name);
    public EventInfo GetEvent(String name, BindingFlags bindingAttr);
    public EventInfo[] GetEvents();
    public EventInfo[] GetEvents(BindingFlags bindingAttr);
    public PropertyInfo GetProperty(String name, BindingFlags bindingAttr);
    public PropertyInfo GetProperty(String name, Type returnType, Type[] types);
    public PropertyInfo GetProperty(String name, Type[] types);
    public PropertyInfo GetProperty(String name, Type returnType);
    public PropertyInfo GetProperty(String name);
    public PropertyInfo[] GetProperties(BindingFlags bindingAttr);
    public PropertyInfo[] GetProperties();
    public Type[] GetNestedTypes();
    public Type GetNestedType(String name);
    public Type[] GetGenericArguments();
}
```


Excerto da classe Type (propriedades)

```
class Type : MemberInfo {  
  
    Type DeclaringType{ get; }  
    Type ReflectedType{ get; }  
    Module Module{ get;}  
    Assembly Assembly{ get; }  
    String FullName{ get; }  
    String Namespace{ get; }  
    Type BaseType{ get; }  
    Boolean IsNested{ get; }  
    Boolean IsNotPublic{ get; }  
    Boolean IsPublic{ get; }  
    Boolean IsLayoutSequential{ get; }  
    Boolean IsClass{ get;}  
    Boolean IsInterface{ get;}  
    Boolean IsValueType{ get;}  
    Boolean IsAbstract{ get; }  
    Boolean IsSealed{ get; }  
    Boolean IsEnum{ get; }  
    Boolean IsSerializable{ get; }  
    Boolean IsArray{ get; }  
    Boolean IsGenericType{ get; }  
    Boolean IsPrimitive{ get; }  
  
}
```

MethodBase (excerto)

```
class MethodBase : MemberInfo {
    public ParameterInfo[] GetParameters();
    public Type[] GetGenericArguments();
    public Object Invoke(Object obj, Object[] parameters);
    Boolean IsOverloaded{ get; }
    Boolean IsGenericMethod{ get; }
    Boolean IsPublic{ get; }
    Boolean IsPrivate{ get; }
    Boolean IsFamily{ get; }
    Boolean IsAssembly{ get; }
    Boolean IsFamilyAndAssembly{ get; }
    Boolean IsFamilyOrAssembly{ get; }
    Boolean IsStatic{ get; }
    Boolean IsFinal{ get; }
    Boolean IsVirtual{ get; }
    Boolean IsHideBySig{ get; }
    Boolean IsAbstract{ get; }
    Boolean IsSpecialName{ get; }
    Boolean IsConstructor{ get; }
    String Name{ get; }
    Type DeclaringType{ get; }
    Type ReflectedType{ get; }
    Module Module{ get; }
}
```

ConstructorInfo e MethodInfo

```
class ConstructorInfo : MethodBase {
    public Object Invoke(Object[] parameters);
    public ParameterInfo[] GetParameters();
    public Type[] GetGenericArguments();
    public Object Invoke(Object obj, Object[] parameters);
}

class MethodInfo : MethodBase {
    public Type[] GetGenericArguments();
    public ParameterInfo[] GetParameters();
    public MethodImplAttributes GetMethodImplementationFlags();
    public Object Invoke(Object obj, Object[] parameters);
    Type ReturnType{ get; }
    ParameterInfo ReturnParameter{ get; }
}
```

EventInfo e PropertyInfo

```
class EventInfo : MemberInfo {  
    public MethodInfo GetAddMethod();  
    public MethodInfo GetRemoveMethod();  
    public MethodInfo GetRaiseMethod();  
    public Void AddEventHandler(Object target, Delegate handler);  
    public Void RemoveEventHandler(Object target, Delegate handler);  
    Type EventHandlerType{ get; }  
    Boolean IsSpecialName{ get; }  
    Boolean IsMulticast{ get; }  
}
```

```
class PropertyInfo : MemberInfo {  
    public ParameterInfo[] GetIndexParameters();  
    public Object GetValue(Object obj, Object[] index);  
    public Void SetValue(Object obj, Object value, Object[] index);  
    public MethodInfo[] GetAccessors();  
    public MethodInfo GetGetMethod();  
    public MethodInfo GetSetMethod();  
    Type PropertyType{ get; }  
    Boolean CanRead{ get; }  
    Boolean CanWrite{ get; }  
}
```

FieldInfo e ParameterInfo

```
class ParameterInfo {
    Type ParameterType{ get; }
    String Name{ get; }

    Int32 Position{ get; }

    Boolean IsIn{ get; }
    Boolean IsOut{ get; }

}

class FieldInfo : MemberInfo {
    public Object GetValue(Object obj);
    public Void SetValue(Object obj, Object value);
    public Type GetType();

    Type FieldType{ get; }
    Boolean IsLiteral{ get; }
    Boolean IsNotSerialized{ get; }
    Boolean IsSpecialName{ get; }

}
```

Introspecção na FCL

```
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;  
    }  
}
```

Introspecção e Geração de Código

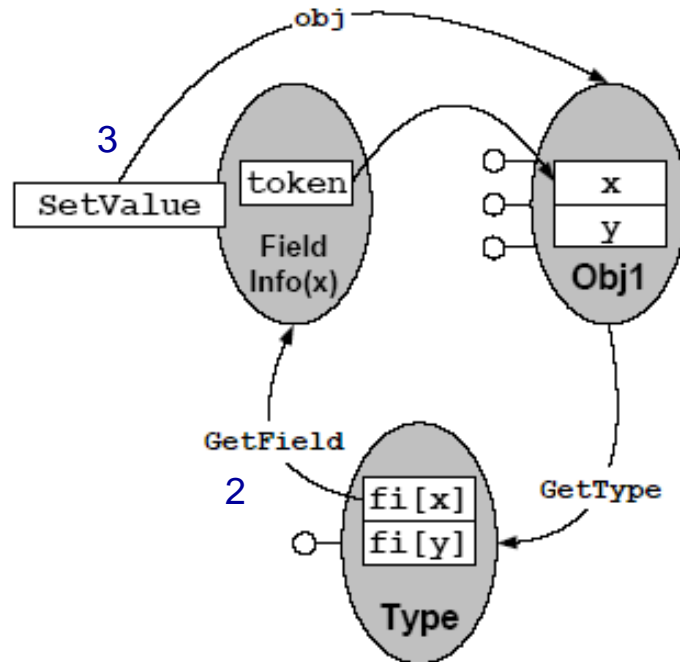
```
public static void GenerateSQL(Object obj) {
    Type type = obj.GetType();
    Console.Write("create table {0} (", type.Name);
    bool needsComma = false;
    foreach (FieldInfo field in type.GetFields()) {
        if (needsComma)
            Console.Write(", ");
        else
            needsComma = true;
        Console.Write("{0} {1}", field.Name, field.FieldType);
    }
    Console.WriteLine(")");
}
```

```
class Pessoa {
    public string nome;
    public string bi;
}
```

create table Pessoa (nome varchar(256), bi varchar(256))

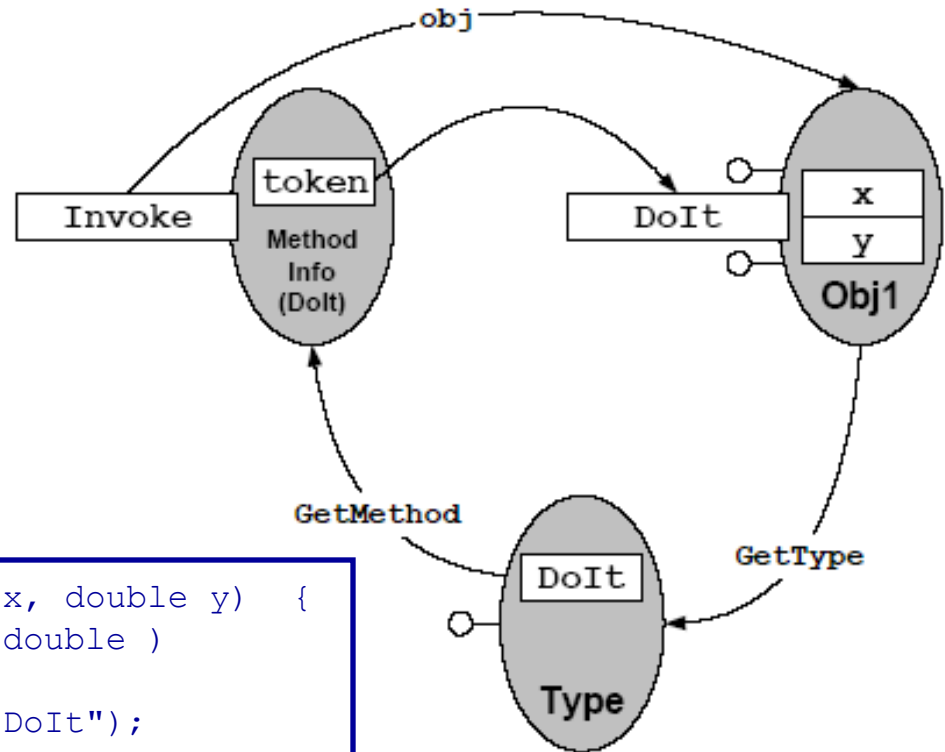
Instrução SQL (Standard Query Language) para a criação de nova tabela em base de dados

Usos de introspecção – afectar campos



```
public static void SetX(Object target, int value) {  
    // Write to field: int x  
    Type type = target.GetType();  
    FieldInfo field = type.GetField("x");  
    field.SetValue(target, value);  
}
```


Usos de introspecção – invocar métodos



```
double CallDoIt(Object target, double x, double y) {  
    // Call method: double Add( double, double )  
    Type type = target.GetType();  
    MethodInfo method = type.GetMethod("DoIt");  
    if( method != null ) {  
        object[] args = { x, y };  
        object result = method.Invoke(target, args);  
        return (double)result;  
    }  
    return(0);  
}
```

Atributos de metadata

- ◆ A maior parte dos elementos de metadata têm um item de 32 bits usado para especificar determinadas características do tipo, campo ou método a que está associado
- ◆ A interpretação destes atributos está fixada na especificação CLI. São exemplos:
 - **initonly** (campo)
 - **beforefieldinit** (tipo)
 - **hidebysig** (método)

Custom attributes

- As linguagens de programação podem mostrar os atributos como modificadores específicos, modificando o nome (Ex: **readonly** (C#) corresponde ao atributo *initonly*, **const** (C#) corresponde ao atributo *literal*)
- Para evitar a proliferação de nomes para representar o mesmo conceito e principalmente para poder criar novos atributos, estendendo a metadata, foi concebido o mecanismo de **custom attribute**. Um **custom attribute** pode ser associado a qualquer elemento de metadata (Assembly, Tipo, Campo, Método, etc.) .
- Em C# um **custom attribute** é usado prefixando o elemento a que está associado com a sintaxe : **[declaração de atributo]**

Exemplo: para informar o sistema de serialização
que um tipo é serializável faz-se (em C#):


```
[Serializable] class A { ... }
```

O tipo A é
serializável

Custom Attributes e invocação de serviços unmanaged

```
[StructLayout(LayoutKind.Sequential, CharSet=CharSet.Auto)]
class OSVERSIONINFO {
    public OSVERSIONINFO() {
        OSVersionInfoSize = (UInt32) Marshal.SizeOf(this);
    }
    public UInt32 OSVersionInfoSize = 0;
    public UInt32 MajorVersion      = 0;
    public UInt32 MinorVersion      = 0;
    public UInt32 BuildNumber       = 0;
    public UInt32 PlatformId        = 0;
    [MarshalAs(UnmanagedType.ByValTStr, SizeConst=128)]
    public String CSDVersion        = null;
}

class MyClass {
    [DllImport("Kernel32", CharSet=CharSet.Auto, SetLastError=true)]
    public static extern Boolean GetVersionEx([In, Out] OSVERSIONINFO version);
}
```



**Existem 266
custom attributes
na FCL (Versão 1.1)**

Custom Attributes (exemplos I)

◆ Custom Attributes usados em Compile Time...

```
namespace Sample {  
    public class TestClass {  
        [CLSCompliant(false)] public int SetValue(UInt32 value);  
        [Obsolete("Use NewMethod instead of OldMethod")]  
        public static void OldMethod()  
        {  
            Console.WriteLine("Hi World");  
        }  
        public static void NewMethod()  
        {  
            Console.WriteLine("Hello World");  
        }  
    }  
}
```

Custom Attributes (exemplos II)

◆ Custom Attributes usados em Load Time...

```
namespace Sample {  
    [StructLayout(LayoutKind.Sequential, CharSet=CharSet.Auto)]  
    public class TestClass {  
        int i1;  
        int i2;  
        string name;  
  
        [MethodImpl(MethodImplOptions.Synchronized)]  
        void f() {...}  
  
        [STAThread ]  
        public static void Main()  
        {  
            ...  
        }  
    }  
}
```

Custom Attributes (exemplos III)

◆ Custom Attributes usados em Run Time

```
using System;  
using Runtime.InteropServices;
```

```
[Serializable]
```

```
class Sample {
```

```
[DllImport("user32.dll")]
```

```
public static extern int  MessageBoxA(int p, string m, string h, int t);
```

```
public static int Main()  {
```

```
    return MessageBoxA(0, "Hello World!",  "DllImport Sample", 0);
```

```
}
```

```
}
```



pseudo-custom attributes

- Certos *custom attributes* são designados por ***pseudo custom attributes***, porque embora sejam usados com a sintaxe de ***custom attribute***, já estão **pré-definidos no CLI**.

System.**NonSerializedAttribute**

System.**SerializableAttribute**

System.Runtime.CompilerServices.**MethodImplAttribute**

System.Runtime.InteropServices.ComImportAttribute

System.Runtime.InteropServices.DllImportAttribute

System.Runtime.InteropServices.FieldOffsetAttribute

System.Runtime.InteropServices.InAttribute

System.Runtime.InteropServices.MarshalAsAttribute

System.Runtime.InteropServices.OptionalAttribute

System.Runtime.InteropServices.OutAttribute

System.Runtime.InteropServices.PreserveSigAttribute

System.Runtime.InteropServices.StructLayoutAttribute

Custom Attributes (utilização)

- ◆ *Custom attributes* genuínos são *strongly typed*. Um *custom attribute* é uma instância de um tipo que deriva directa ou indirectamente de `Attribute`. Sejam as seguintes definições em C#:

- `public sealed class TestedAttribute : Attribute{}`
- `public sealed class DocumentedAttribute : Attribute{}`

Utilização de Custom
Attributes ()

```
public sealed class MyCode {  
    [ TestedAttribute ]  
    [ DocumentedAttribute ]  
    static void f() {}  
  
    [ Tested ]  
    static void g() {}  
  
    [ Tested, Documented ]  
    static void h() {}  
}
```

pode-se omitir em
C# o prefixo
Attribute

Podem ser feitas
várias declarações
em simultâneo

Custom Attributes


(resolver ambiguidades na utilização)

```
[assembly: Red ]
[module: Green ]
[class: Blue ] } Atributos da classe
[ Yellow ]
public sealed class Widget
{
    [return: Cyan ]
    [method: Magenta ] } Atributos do Método
    [ Black ]
    public int Splat() {}
}
```

Definição de Custom Attributes

```
[AttributeUsage(AttributeTargets.Method)]
public sealed class DocumentedAttribute : Attribute {
    public DocumentedAttribute() { }
    public DocumentedAttribute(string w) { Writer = w; }
    public string Writer;
    public int    WordCount;
    public bool   Reviewed;
}
```

Ao definir o construtor de instância, os campos e as propriedades de um derivado de `Attribute`, apenas é possível usar um pequeno subconjunto de tipos, especificamente: ***Boolean, Char, Byte, SByte, Int16, UInt16, Int32, UInt32, Int64, UInt64, Single, Double, String, Type*** e tipos enumerados. É também possível utilizar *arrays* unidimensionais (vectores) de qualquer destes tipos.



```
public sealed class MyCode {
    [ Documented("Don Box",
        WordCount = 42) ]
    static void f() {}

    [ Documented(WordCount = 42,
        Reviewed = false) ]
    static void g() {}

    [ Documented(Writer = "Don Box",
        Reviewed = true) ]
    static void h() {}
}
```

restrições na utilização de atributos (exemplo)

```
[AttributeUsage(AttributeTargets.Method | AttributeTargets.Property , AllowMultiple = true, Inherited = true)]
```

```
public class AuthorAttribute: Attribute {  
    private string FamilyName;  
    private string GivenName;  
    public AuthorAttribute(string FamilyName) {  
        this.FamilyName = FamilyName;  
    }  
    public override String ToString() {  
        return String.Format("Author: {0} {1}", Family, Given);  
    }  
    public string Family {  
        get { return FamilyName; }  
        set { FamilyName = value; }  
    }  
    public string Given {  
        get {  
            return GivenName;  
        }  
        set {  
            GivenName = value;  
        }  
    }  
}
```

```
namespace System {  
    [AttributeUsage(AttributeTargets.Enum, Inherited= false)]  
    public class FlagsAttribute : Attribute {  
        public FlagsAttribute() { }  
    }  
}
```

Limitar a utilização de *custom attributes* a certos elementos. O enumerado **AttributeTargets**

Nome	Descrição
All	Attribute can be applied to any application element.
Assembly	Attribute can be applied to an assembly.
Class	Attribute can be applied to a class.
Constructor	Attribute can be applied to a constructor.
Delegate	Attribute can be applied to a delegate.
Enum	Attribute can be applied to an enumeration.
Event	Attribute can be applied to an event.
Field	Attribute can be applied to a field.
Interface	Attribute can be applied to an interface.
Method	Attribute can be applied to a method.
Module	Attribute can be applied to a module.
Parameter	Attribute can be applied to a parameter.
Property	Attribute can be applied to a property.
ReturnValue	Attribute can be applied to a return value.
Struct	Attribute can be applied to a value type.

AttributeUsageAttribute

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
[Serializable]
public sealed class AttributeUsageAttribute : Attribute {
    internal AttributeTargets m_attributeTarget = AttributeTargets.All;
    internal Boolean m_allowMultiple = false;
    internal Boolean m_inherited = true;
    public AttributeUsageAttribute(AttributeTargets validOn) {
        m_attributeTarget = validOn;
    }
    public AttributeTargets ValidOn {
        get { return m_attributeTarget; }
    }
    public Boolean AllowMultiple {
        get { return m_allowMultiple; }
        set { m_allowMultiple = value; }
    }
    public Boolean Inherited {
        get { return m_inherited; }
        set { m_inherited = value; }
    }
}
```

A classe abstracta Attribute

Public Methods

Static <u>GetCustomAttribute</u>	Overloaded. Retrieves a custom attribute of a specified type applied to a specified member of a class.
Static <u>GetCustomAttributes</u>	Overloaded. Retrieves an array of the custom attributes of a specified type applied to a specified member of a class.
<u>IsDefaultAttribute</u>	When overridden in a derived class, returns an indication whether the value of this instance is the default value for the derived class.
Static <u>IsDefined</u>	Overloaded. Determines whether any custom attributes of a specified type are applied to a specified member of a class.
<u>Match</u>	When overridden in a derived class, returns a value indicating whether this instance equals a specified object.

Forma
especializada de
comparação

Custom Attributes

Verificar a
existência de
**Custom
Attributes**

```
namespace Reflection {
public interface ICustomAttributeProvider {
    // test for presence or absence of attribute of type attType
    bool IsDefined(Type attType, bool inherited);

    // return all attributes that are compatible with attType
    object[] GetCustomAttributes(Type attType,
                                bool inherited);

    // return all attributes irrespective of type
    object[] GetCustomAttributes(bool inherited);
}
}
```

```
using System;
using Reflection;
```

```
public sealed class Utils {
    static void DisplayMethodStatus(Type type) {
        foreach (MethodInfo m in type.GetMethods()) {
            Console.WriteLine("{0} : ", m.Name);
            // check the doc'ed attribute
            if (m.IsDefined(typeof(DocumentedAttribute), true))
                Console.WriteLine("Documented");
            else
                Console.WriteLine("Undocumented");
            // check the tested attribute
            if (m.IsDefined(typeof(TestedAttribute), true))
                Console.WriteLine(" - OK");
            else
                Console.WriteLine(" - Broken");
        }
    }
}
```

**MemberInfo, ParameterInfo,
Assembly, Module e Type,**
entre outros, suportam a
interface:
ICustomAttributeProvider