Reflexão e Anotações (*Custom Attributes*)

Centro de Cálculo
Instituto Superior de Engenharia de Lisboa

F. Miguel Carvalho (mcarvalho@cc.isel.ipl.pt)
Nuno Leite (nleite@cc.isel.ipl.pt)

Agenda

- Reflexão e Introspecção
- Anotações (*Custom Attributes*)

Introspecção <> Reflexão

- **Reflexão**, o que é?
  - Disponibilização, via API, de informação de tipo (*metadata*) em tempo de execução.
  - Em .Net esta API é composta pelas classes do *namespace* `System.Reflection`.
- **Introspecção**, o que é?
  - Realização de código genérico que tira partido da existência de **reflexão**.

Linguagem C# - Reflexão

- Informação completa de tipo (*metadata*) em tempo de execução

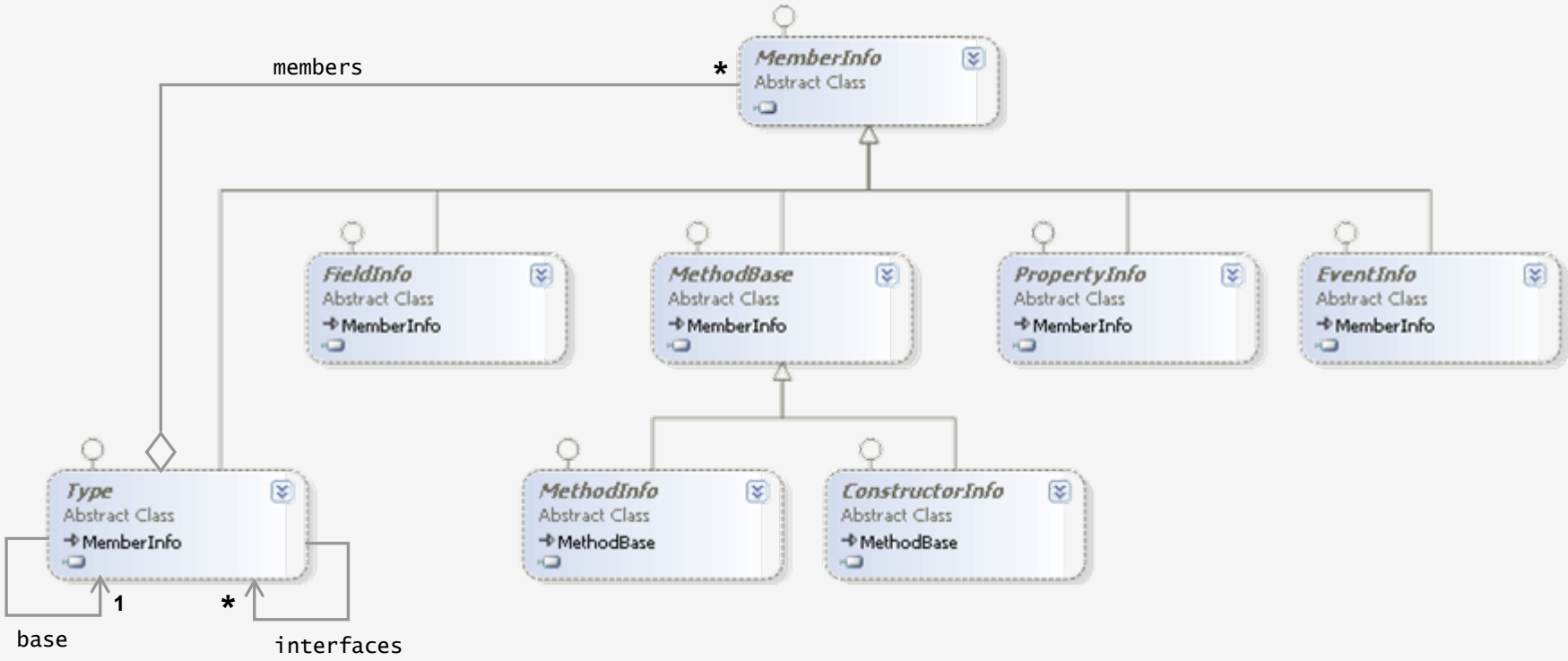
- Permite a realização de verificações para detecção de erros de codificação, aumentando a robustez do programa, como seja validade das conversões:
  - E.g.: **`IsSubclassOf`** **e** **`IsAssignableFrom`**

- Essa informação é acessível programaticamente via a API de Reflexão:
  - Classe **`System.Type`** e classes do *namespace* **`System.Reflection`**.

- A técnica de realização de código genérico recorrendo a essa informação designa-se introspecção.

Hierarquia de classes de reflexão (simplificada)

Os seus objectos são representantes de Tipos e Membros.



Assembly

Classe `Assembly`:

- método estático `Assembly LoadFrom(string path)`
  - Carrega um *assembly* dado o seu nome ou caminho.

- método de instância `Type[] Assembly.GetExportedTypes()`
  - Devolve os tipos públicos definidos por um *assembly*

```
Assembly a = Assembly.LoadFrom(fileName);
Type[] types = a.GetExportedTypes();
foreach (Type t in types) {
   // Do something
   // ...
}
```

Assembly → Type

- `System.Type` é o ponto de partida para a introspecção dos membros.
- `System.Type` é um tipo base abstracto derivado de `MemberInfo`.
  - porque um tipo também pode ser um membro de outro tipo - *nested*;
- `System.RuntimeType`
  - tipo interno da FCL, derivado de `System.Type`;
  - a primeira vez que um tipo é ACEDIDO, o CLR constrói uma instância de `RuntimeType` com informação desse tipo;
  - O método de instância `Object.GetType()`, determina o tipo de uma instância e retorna uma referência para o seu `RuntimeType`;

  ⇒ pode ser usado o operador == para verificar se dois objectos são instâncias do mesmo tipo.

- ATENÇÃO à diferença entre testar com `is` e o `GetType`.

Assembly → Type...

- *Query* sobre `Type`. Propriedades:
  - isPublic, isSealed, isAbstract, isClass, isValueType;
  - Assembly, AssemblyQualifiedName, FullName;
  - BaseType.

Criação de instâncias:

- Classe `Activator` → método estático `Object CreateInstance(...)`:
  - recebe uma referência do tipo ou uma *String*;
  - se for passada *string* tem que ser uma *assembly-qualified string* (e.g. "`tipo, assembly, version=2.0.0.0`");
  - pode receber um conjunto de argumentos para o construtor;

- **ATENÇÃO: Para instâncias de *arrays* e *delegates* usar os métodos `CreateInstance` específicos das classes `Array` e `Delegate`.**

Assembly → Type → MemberInfo

- `MemberInfo` - Classe abstracta com um conjunto de propriedades comuns a todos os membros:
  - `String Name`,
  - `MemberTypes MemberType`,
  - `Type DeclaringType` e `Type ReflectedType`.

- Obter os membros definidos por um tipo:
  - `MemberInfo[] System.Type.GetMembers(BindingFlags bf)`
- ALTERNATIVA:
  - `GetConstructors`, `GetMethods`, `GerProperties`, `GetEvents`

Assembly → Type → MemberInfo ←– FieldInfo

- Obter o valor de um campo:
  - `Object GetValue(Object obj)`

    `obj` – objecto sobre o qual é obtido o valor do campo.

Introspecção: exemplo

```
using System;
using System.Reflection;
public class Inspector {
  public static void Inspect(object obj) {

    // Obter a instância de System.Type
    // que representa o tipo do objecto
    Type typ = obj.GetType();

    // Mostrar nome do tipo
    Console.Write(typ.Name + ": ");

    // Obter conjunto de campos
    FieldInfo[] flds = typ.GetFields(
      BindingFlags.Instance | BindingFlags.Public | BindingFlags.NonPublic);

    // Percorrer conjunto de campos
    foreach (FieldInfo fld in flds) {
      // Mostrar valor de cada campo
      Console.Write("({0} : {1}) ", fld.Name, fld.GetValue(obj));
    }
    Console.WriteLine();
  }
}
```

Assembly → Type → MemberInfo ◄– MethodInfo

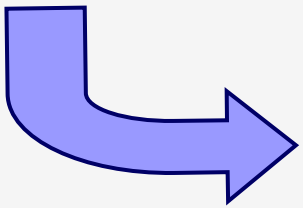
- Consultar o protótipo de um método:
  - `Type ReturnType`
  - `ParameterInfo[] GetParameters()`

    Propriedade `ParameterType` de `ParameterInfo` dá o tipo do parâmetro.

- Invocação do método:
  - `Object Invoke( Object obj, Object[] parameters)`

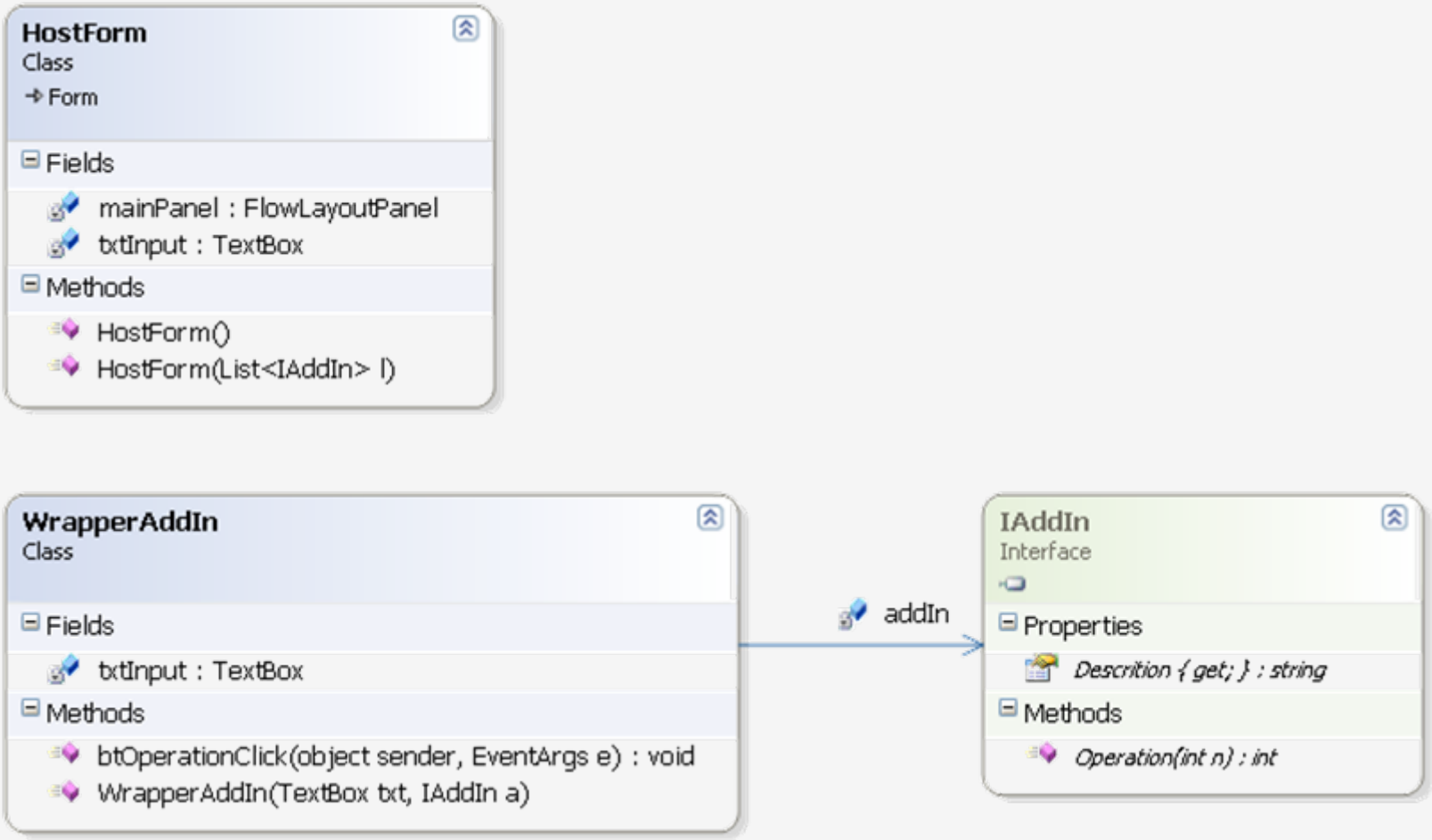
    ATENÇÃO: se possível evitar => custo de desempenho.

```
Object ai = Activator.CreateInstance(t);
MethodInfo op = t.GetMethod("Operation");
op.Invoke(ai, new Object[] {n});
```



```
if (typeof(IAddIn).IsAssignableFrom(t)) {
  IAddIn ai = (IAddIn)Activator.CreateInstance(t);
  ai.Operation(n);
}
```

Introspecção: exemplo com AddIns



Assembly → Type → MemberInfo ◄– EventInfo

- propriedade `Type EventHandlerType`
  tipo *delegate* do evento.

- método `AddEventHandler(Object target, Delegate handler)`
  para adicionar um *handler* ao evento do objecto `target`.

- método `RemoveEventHandler( Object target, Delegate handler )`
  para remover um *handler* ao evento do objecto `target`.

Introspecção: exemplo

O método `TraceAllEvents` faz com que o disparo de qualquer evento de ‘o‘ com tipo `EventHandler`, produza uma mensagem com o nome desse evento.

```
public class Tracer {
  string eventName;
  public Tracer(string en){
    eventName = en;
  }
  public void TraceEvent(object sender, EventArgs ea) {
    Console.WriteLine(string.Format("{0} event fired", eventName));
  }
}
class Program {
 public static void TraceAllEvents(object o) {
    Type t = o.GetType();
    EventInfo[] events = t.GetEvents();
    foreach (EventInfo ei in events) {
      if (ei.EventHandlerType == typeof(EventHandler)) {
        Tracer tracer = new Tracer(ei.Name);
        ei.AddEventHandler(o, new EventHandler(tracer.TraceEvent));
      }
    }
 }
```

Agenda

- Reflexão e Introspecção
- Anotações (*Custom Attributes*)

Common Type System - Atributos

- Informação completa de tipo descrita em *metadata* (representação intermédia de tipos) de acordo com regras do *Common Type System*

- Tipos e membros caracterizados por atributos pré-definidos
  - `public`, `private`, `static`, `abstract`, ...
  - representado internamente com um valor a 32 bits

- Caracterização adicional através de outros atributos
  - mecanismo de extensão de *metadata*
  - atributos especializados (*custom attributes*)

- É possível aplicar atributos a
  - *assemblies*, módulos, tipos, propriedades, eventos, campos, métodos e parâmetros e valores de retorno

Custom Attributes… utilização

pode-se omitir em C# o sufixo **Attribute**

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

Podem ser feitas várias declarações em simultâneo

```
[assembly: Red]
[module: Green]
[type: Blue]
[Yellow]
public sealed class Widget {
   [method: Magenta]
   [Black]
   public int Splat() { }
}
```

Atributos da classe

Atributos do Método

Custom Attributes… definição

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

```
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() { }
}
```

Ao definir o construtor de instância do tipo atributo, os campos e as propriedades apenas é possível usar um pequeno subconjunto de tipos, especificamente: ***Boolean***, ***Char***, ***Byte***, ***SByte***, ***Int16***, ***UInt16***, ***Int32***, ***UInt32***, ***Int64***, ***UInt64***, ***Single***, ***Double***, ***String***, ***Type***, ***Object*** **e tipos enumerados**. É também possível utilizar *arrays* com uma dimensão e índices baseados em zero de qualquer um destes tipos.

Custom Attributes

- Um atributo especializado representa um aspecto que não tem representação pré-definida em *metadata*

- Atributos especializados são instâncias de tipos
  - CLS obriga a que sejam tipos derivados de `System.Attribute`

- Resultado da aplicação de um atributo especializado
  - informação seriada para *metadata*

- Atributos podem ser consultados em tempo de execução
  - `bool IsDefined( Type attributeType, bool inherit )`
    saber se um elemento está qualificado com um determinado tipo de atributo.
  - `Object[] GetCustomAttributes(Type att, bool inherit)`
    obter todos os atributos especializados associados a um elemento.

Custom Attributes: exemplo

```
using System;
using System.Reflection;

public class InspectableAttribute : Attribute { }

public class Inspector {

   public static void Inspect(object obj) {

      // ...

      // Percorrer conjunto de campos
      foreach (FieldInfo fld in flds) {
         // Mostrar valor de cada campo, se for inspeccionável
         if (fld.IsDefined(typeof(InspectableAttribute), false)) {
            Console.WriteLine("   {0} : {1}", fld.Name, fld.GetValue(obj));
         }
      }
   }
}
```

Restrições na utilização de atributos

- Aplicar o atributo `AttributeUsageAttribute` para indicar ao compilador quais os tipos de TARGET aplicáveis ao atributo
  - propriedades:

    ```
    bool AllowMultiple
    bool Inherited
    AttributeTargets ValidOn;
    ```

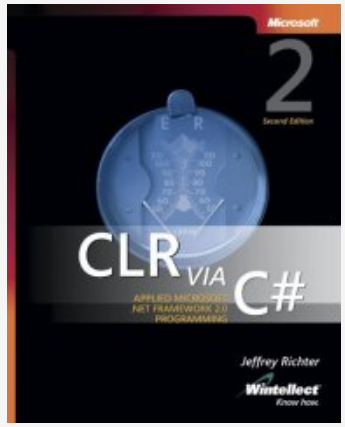
  - `AllowMultiple`:

    por omissão `false` => o atributo não pode ser aplicado mais que uma vez ao mesmo *target*.

```
[AttributeUsage(AttributeTargets.Field, AllowMultiple = false)]
public class InspectableAttribute:Attribute{
}
```

Bibliografia recomendada

Jeffrey Richter, “CLR via C#, Second Edition”,
Microsoft Press; 2nd edition, 2006



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