Microsoft .net

C# 3.0

Consolidations

Key Consulting

Agenda

- Reflexion
- XML
- Garbage Collector

Agenda

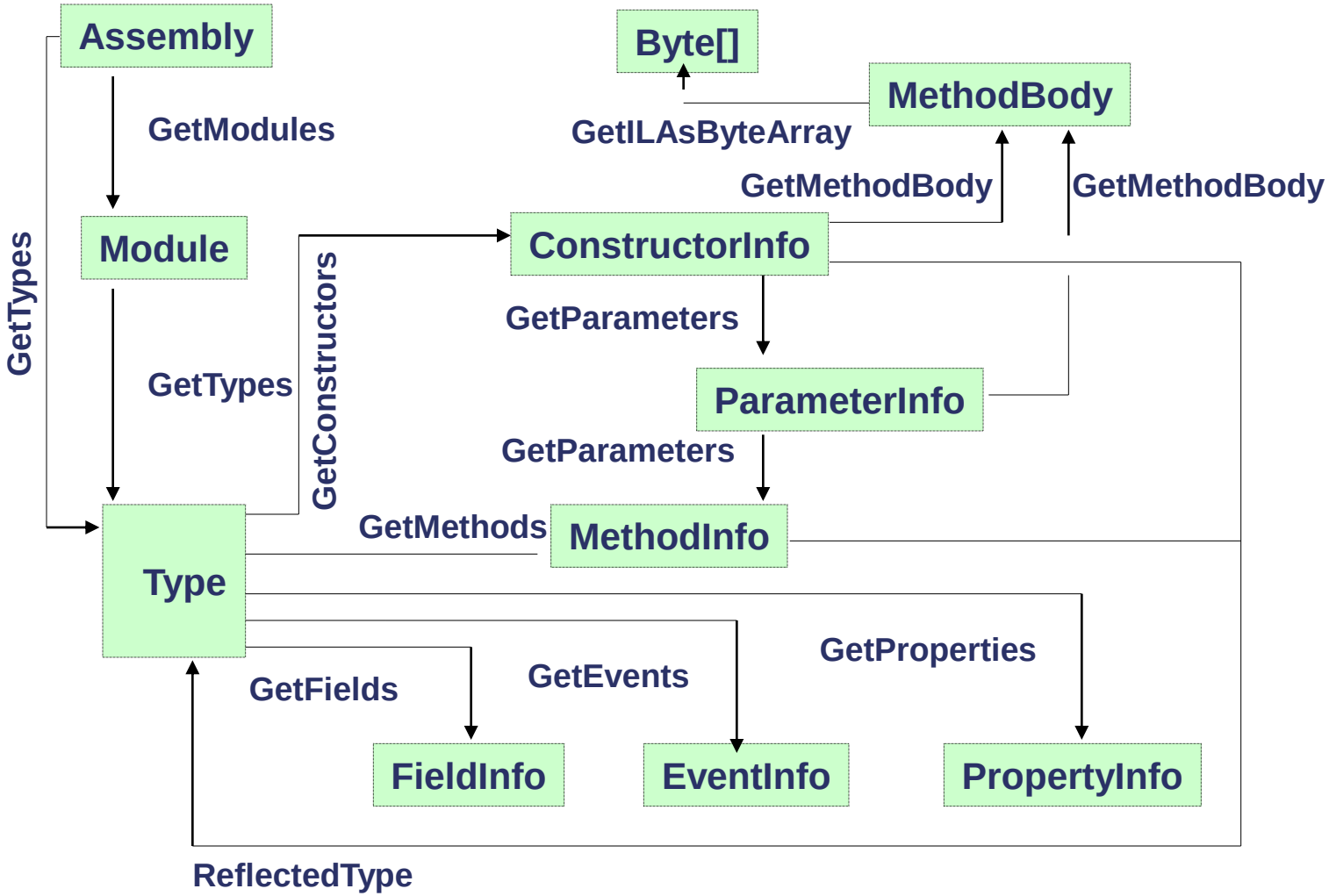
- Reflection
- XML
- Garbage Collector

Reflection : Définition

« Accéder aux meta-données des assemblages »

- **Utilisés pour :**
  - La découverte d'un assemblage à l'exécution
  - Lorsque l'on a un *lien tardif* vers un assemblage
  - Lorsque l'on souhaite utiliser les attributs
  - Pour créer une instance de type à partir d'une définition
  - Accéder aux membres non public
  - …

Reflection : Lecture

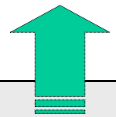


Reflection : Lecture des attributs

```
[AttributeUsage(AttributeTargets.Class)]
class MyAttribute : System.Attribute {
  private int _value;
  public int Value {
    get { return _value; }
    set { _value = value; }
  } }
```

```
[MyAttribute(Value = 10)]
public class Foo { }
```

10



```
Foo foo = new Foo();
object[] attributes =
   foo.GetType().GetCustomAttributes(typeof(MyAttribute),false);
if (attributes.GetLength(0)!=0) {
 System.Console.WriteLine(((MyAttribute)attributes[0]).Value);
}
```

Reflection : Appel à une méthode privée

```
public class Foo {
 private void PrivateMethod() {
   System.Console.WriteLine("PrivateMethod");
 }
}
```

```
Foo foo = new Foo();
foo.GetType().InvokeMember("PrivateMethod",
   BindingFlags.Instance |
   BindingFlags.NonPublic |
   BindingFlags.InvokeMethod, null, foo,
   null);
```

BindingFlags précise le domaine d'action de InvokeMethod

PrivateMethod

Reflection : Lien avec une classe

Lien entre la couche logicielle qui utilise la classe et celle qui en possède la définition

- **Liens précoces :**
  - Méthode statique
  - Méthode classe non virtuelle …
- **Liens dynamiques :**
  - Méthodes virtuelles
- **Liens tardifs :**
  - Créés après la compilation …

Reflection : Création d'une classe

- **4 méthodes pour créer une classe**
  - `AppDomain.CreateInstanceAndUnwrap`
  - `Activator`
  - `ConstructorInfo`
  - `Type.InvokeMember`
- **Cas particuliers**
  - Tableau : `Array.CreateInstance`
  - Delegate : `Delegate.CreateInstance`

Reflection : Création d'une classe (exemple)

```
class ClassLibrary.Foo {}
```

Solution 1

```
object obj =
AppDomain.CurrentDomain.CreateInstanceFromAndUnwrap
                    ("ClassLibrary.dll", "ClassLibrary.Foo");
```

Solution 2

```
Assembly assembly =
        Assembly.LoadFile(@"c:\ClassLibrary1.dll");
Type type = assembly.GetType("ClassLibrary1.Foo");
ConstructorInfo constrcutorInfo =
        type.GetConstructor(new Type[0]);
object obj = constrcutorInfo.Invoke(new object[0]);
```

Reflection : Création d'une classe (exemple)

Solution 3

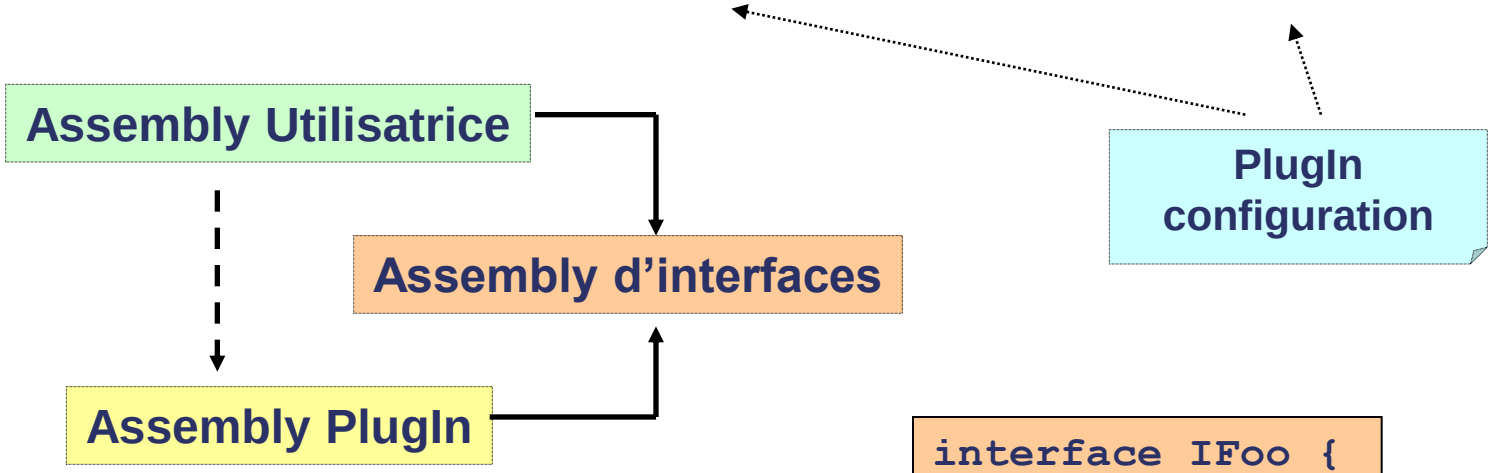
```
Assembly assembly = Assembly.LoadFile(@"c:\ClassLibrary1.dll");
Type type = assembly.GetType("ClassLibrary1.Foo");
object obj = Activator.CreateInstance(type);
```

Solution 4

```
Assembly assembly = Assembly.LoadFile(@"c:\ClassLibrary1.dll");
Type type = assembly.GetType("ClassLibrary1.Foo");
object obj = type.InvokeMember(
        null,BindingFlags.CreateInstance,null,null,new object[0]);
```

Reflection : PlugIn

```
IFoo obj =
AppDomain.CurrentDomain.CreateInstanceFromAndUnwrap
            ("ClassLibrary.dll", "ClassLibrary.Foo");
```




```
interface IFoo {
}
```

```
class Foo : IFoo {
}
```

Reflection : Générique

- La « reflection » permet de créer des types génériques fermés à la volée :

```
class Generique<T> {
}
```

```
class Foo {
}
```

```
Type typeOuvert = typeof (Generique<>);
Type typeFerme = typeOuvert.MakeGenericType(typeof(Foo));
object obj = Activator.CreateInstance(typeFerme);
```

Reflection : TypeDescriptor

- **TypeDescriptor :**
  - Donne des informations sur
    - un objet
    - un type
  - Plusieurs types d'informations;
    - Properties
    - Events
    - Attributs
    - Converter
    - Editor
    - …

```
foreach (PropertyDescriptor pD in
   TypeDescriptor.GetProperties(o))
{
  Console.WriteLine("Property  P "
                          + pD.Name);
}
```

Reflection : TypeDescriptor

- **Un TypeDescriptor peut être associé à un « Type » ou un « Objet »**
  - Ex :

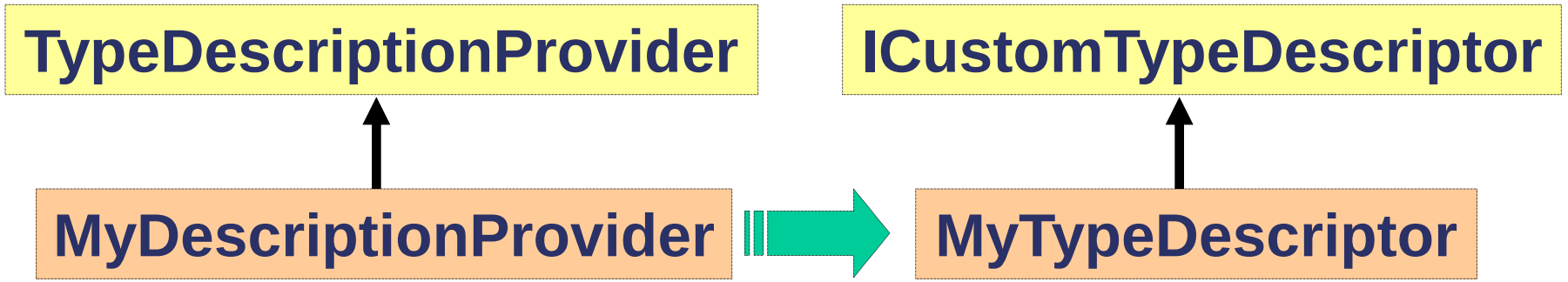
```
TypeDescriptor.GetProperties(o)
TypeDescriptor.GetProperties(typeof(o))
```

- **Personalisation**

```
public static void AddProvider(TypeDescriptionProvider provider,
   object instance);
public static void AddProvider(TypeDescriptionProvider provider,
   Type type);
```

Reflection : TypeDescriptor

Créer son propre TypeDescriptor :



- Ne change pas les « meta-datas »
- Permet de « mentir » sur la description d'un Type ou d'un objet

Agenda

- Reflexion
- XML
- Garbage Collector

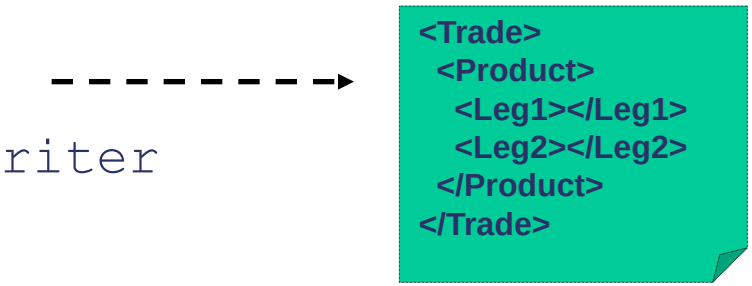
Xml : Overview des technologies

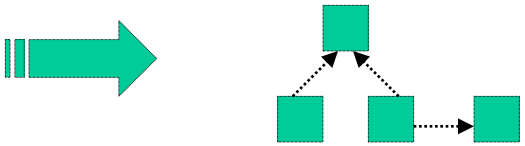
- **XML**
  - Langage à balise de description de données
  - « XML bien formé »
- **XSD**
  - Schéma de validation d'un document
  - Décrit en XML
  - « XML valide »
- **XPath**
  - Langage pour accéder aux données d'un document XML
- **XSLT**
  - Document de transformation d'un document XML en un autre document texte
  - Décrit en XML

Xml : Manipulation d'un document XML

- **Manipulation d'un document XML**

  - A base de curseur
    - `XmlReader` / `XmlWriter`
  - A base d'arbre
    - `XmlDocument`
  - Sérialisation
    - `XmlSerializer`





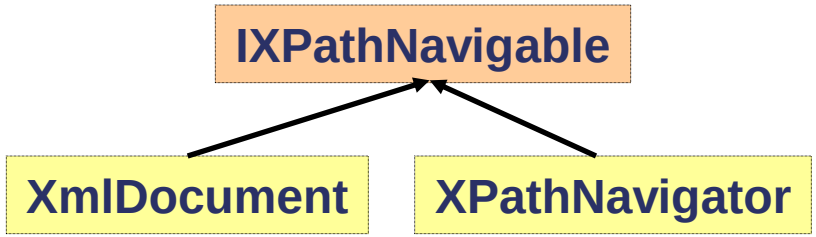
Xml : XmlReader / XmlWriter

- **Sont des classes abstraites**
  - Permet la lecture et l'écriture d'un document XML
  - Exemple de classes concrètes :
    - XmlTextWriter / XmlTextReader
    - Flot de données Textuel

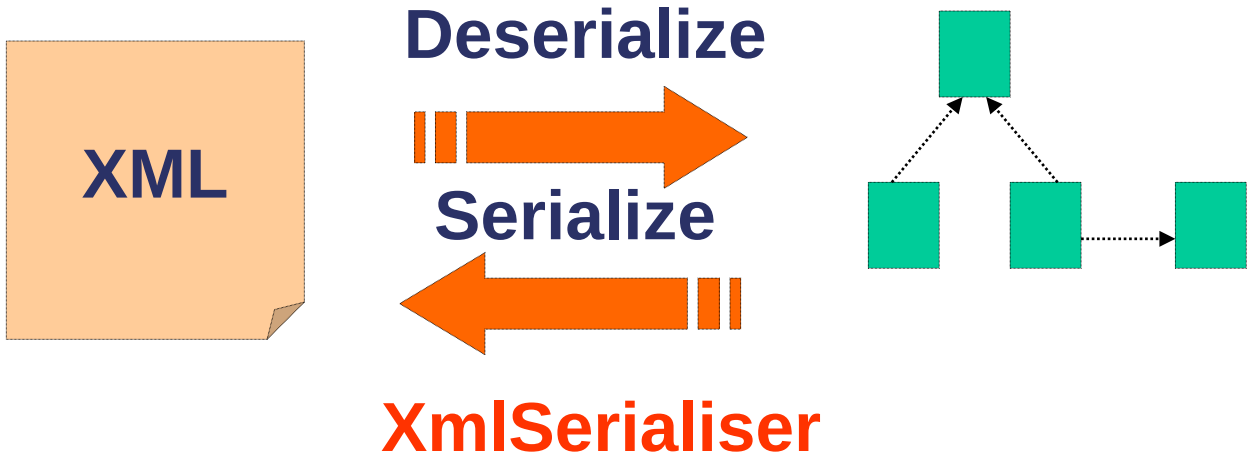
```
XmlTextReader tr = new XmlTextReader("XmlFile1.xml");
while (tr.Read()) {
   System.Console.WriteLine(
        "Nom({0}),Type({1}),Value({2})",
        tr.Name,tr.NodeType,tr.Value);
}
```

Xml : XmlDocument et XPath

- **XmlDocument et XmlNode**
  - SelectSingleNode
  - SelectNodes → **XmlNodeList**
- **XPathNavigator** → **XPathNodeIterator**
  - Select



Xml : Sérialisation



XmlSerialiser

- **Les attributs pour la sérialisation**
  - XmlRoot, XmlElement, XmlAttribute …
  - XmlIgnore

Agenda

- Reflexion
- XML
- Garbage Collector

Garbage Collector

- **Le Garbage Collector « manage » le tas géré**
  - Tout objet de type référence est stocké dans le « tas géré »
- **1 Garbage Collector par processus**
- **Le GC s'occupe de supprimer les références inutiles :**
  - Une instance de type qui n'est plus référencée est inaccessible

ATTENTION !

Le Garbage Collector n'évite pas la fuite mémoire

Garbage Collector : Générations

« Plus un objet est récent plus son espérance de vie est courte »

- **Le GC possède 3 Générations : 0 , 1 , 2**
  - Une instance qui survit à une collecte saute de génération

« Les collectes des générations basses sont plus fréquentes que celles des générations élevées »

Garbage Collector

Etapes du GC

- **Identification des éléments racines de « l'arbre »**
  - Eléments statiques, de la pile …
- **Fabriquer l'arbre de référence**
  - Marquer ceux qui sont encore référencé
- **Désalocation des éléments inactifs**
  - Les « Finalizeur » sont appelés dans un Thread à part
- **Défragmentation du tas**
- **Recalcul des références**

Agir sur le GC

- **GC.Collect(), GC.Collect(int generation)**
  - Déclenche une collecte du ramasse-miettes
- **GC.WaitingForPendingFinalizers()**
  - Suspend le thread courant jusqu'à la fin de tout les « Finalizeur »
- **GC.SupressFinalize(object o)**
  - Enlève l'objet de la liste des objets à « Finaliser »
- **GC.ReRegisterForFinalizer(object o)**

Les références faibles (weak reference)

Un objet référencé par une référence faible peut être collectée par le GC

```
Object o = new object();
System.WeakReference wo = new System.WeakReference(obj);
obj = null;
// le GC peut collecter l'objet
obj = wo.Target; // création d'une référence forte
If (obj == null) {
 // l'objet a été collecté
} else  {
 // l'objet est encore utilisable
}
```

Finalizeur

Syntaxe :

```
public class Foo {
    ~Foo() {}
  }
```

N’existe pas sur les structures

Les « Finalizeurs » sont :

- **Utilisés pour libérer de la mémoire non gérée,**
  - Ne pas appeler d’objets géré
- **Appelés par le GC durant une « collecte »,**
  - Dans un ordre imprévisble
- **Appelés dans un thread à part.**

Dispose Pattern

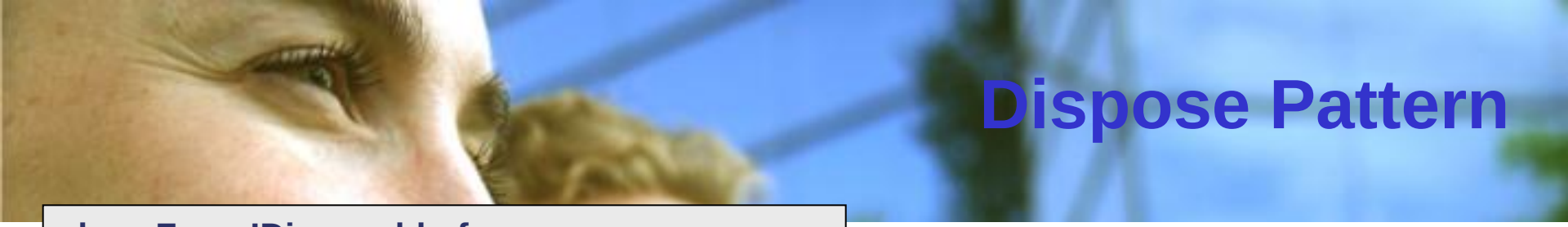
- Design Pattern qui ne change en rien le comportement du GC

```
public interface IDisposable {
        void Dispose();
  }
```

- La méthode Dispose doit être appelée dans une clause Finally
- C# offre une syntaxe simplifiée :

```
A a = new A();
using (a)  {
  ....
}
```

```
A a = new A();
try  {  .... }
Finally {
   a.Dispose();
}
```



Dispose Pattern

```
class Foo : IDisposable {
        private bool m_bDisposed = false;
        public void Dispose() {
            Dispose(true);
            System.GC.SuppressFinalize(this);
        }
        ~Foo() {
            Dispose(false);
        }
```

```
protected virtual void Dispose(bool managedRes)
    {
            if ( ! m_bDisposed ) {
                m_bDisposed = true;
                // ressources non gérées
                if (managedRes) {
                // ressources gérées
                }
            }
        }
    }
```