TP Réflexion

La réflexion est la capacité d'un programme à examiner sa structure interne (introspection) et à la modifier (intercession). La réflexion permet d'inspecter dynamiquement le contenu d'*assemblages*, d'en lire les types, de créer des instances de ces types durant l'exécution du programme et d'appeler leurs méthodes ou champs dynamiquement. On peut même en .Net générer facilement du code IL à l'exécution. Il existe d'autres mécanismes s'approchant de la réflexion utilisés avant .NET, comme RTTI (*Run-Time Type Identification*) en C/C++ ou, dans une certaine mesure, l'interface *IDispatch* pour les composants COM.

Le mécanisme de réflexion au sein de C#

Offrant aux programmes des outils d'introspection au sein de leurs propres méthodes, la réflexion est implémentée dans .NET de telle sorte que le développeur puisse la maîtriser sans en subir la complexité.

On peut définir la réflexion comme le fait de récupérer les attributs d'un objet ou de sa classe de manière dynamique.

Ces attributs sont des métadonnées, stockées dans l'assemblage d'une application, se rapportant aux éléments du code, et permettent de préciser les relations au sein du code. Ainsi, les attributs forment des informations sur ces éléments, informations utilisables par le développeur, comme une description ou un usage préférable. .NET met ainsi à disposition nombre d'attributs standards, mais il est possible d'en créer d'autres. Outre le développeur, le CLR (Common Language Runtime, le coeur de .NET) lui-même peut profiter de ces attributs à l'exécution même du programme, ce qui lui permettra d'agir, à partir de ces informations, sur les agissements des éléments "marqués". Enfin, le troisième utilisateur potentiel de ces attributs est le compilateur lui-même, qui peut par leur biais recevoir des instructions de compilation spécifiques. En définitive, ces attributs permettent aux développeurs d'utiliser le langage de la manière la plus efficace pour eux, et en quelque sorte de l'améliorer sans réellement y toucher.

Ces attributs peuvent donc être lus et modifiés pendant l'exécution de l'application, via le mécanisme nommé "réflexion". Ce mécanisme permet au programme de travailler sur lui-même, et d'en savoir plus au sujet des classes utilisées. De fait, c'est une fonctionnalité assez puissante, avec la complexité qui va avec : manipuler dynamiquement les éléments internes d'un programme ne peut pas être fait à la légère, mais autorise des utilisations très élaborées du langage comme de .NET : analyser les capacités des classes, accéder à certaines méthodes, invoquer de nouvelles méthodes...

En pratique

Avec C#, les informations contenues dans les métadonnées sont accessibles via les méthodes des classes proposées par l'espace de nom System.Reflection, auquel s'ajoute la classe System.Type. L'API de réflexion propose de nombreuses manières de lancer la réflexion sur un objet ou une méthode. Par exemple, l'objet Type propose les méthodes GetType, GetProperties, GetMethods, GetConstructors, GetInterface...
De leur côté, les classes offertes par System.Reflection permettent de cibler précisément ce sur quoi le développeur veut agir : Assembly, AssemblyFileVersionAttribute, ConstructorInfo, ManifestResourceInfo...
En fait, à chaque type d'élément d'un assemblage correspond une classe de réflexion, ce qui autorise des actions très détaillées sur les éléments du code.
L'exemple canonique du mécanisme de réflexion reste la récupération des attributs d'un type, au moyen de la classe System.Type. Le programme suivant n'a pas d'autres fonctions que d'afficher dans la console ses propres attributs, ce qui montre combien, malgré la complexité de la réflexion, son usage au sein de .NET a été grandement facilité.

```
using System;
using System.Reflection;

class TestReflexion {
  public static void Main() {
    foreach(Assembly asm in AppDomain.CurrentDomain.GetAssemblies() ) {
      if (asm.GetName().Name != "mscorlib") {
        foreach (Type ty in asm.GetTypes() ) {
          Console.WriteLine("Classe : " + ty);
          foreach (MethodInfo mi in ty.GetMethods() ) {
            Console.WriteLine(" Méthode   : " + mi);
            foreach (ParameterInfo pi in mi.GetParameters() )
              Console.WriteLine(" Paramètre : " + pi.GetType() );
          }
        }
      }
    }
  }
}
```


Créez un projet vide.
Créez une classe de test avec une méthode hello qui écrit hello world sur la console.
Créez un main qui instancie un objet de la classe et qui appelle la méthode hello.
Affichez de manière introspective les méthodes de la classe hello et leurs paramètres.
Affichez uniquement les méthodes qui ne prennent pas de paramètre.

Développez une classe container générique Pair avec des contraintes de type sur les deux objets qu'elle contient.

C# 2.0 : présentation des Generics

Les types génériques existaient déjà du temps du langage C++ (avec la bibliothèque générique STL par exemple), sous le nom de *templates* (modèles). Le concept n'est donc pas nouveau, mais le fait de l'implanter directement dans la langage plutôt que de laisser sa réalisation aux bilbiothèques externes, peut en rendre l'utilisation plus pratique et plus rapide. On peut aussi trouver des formes de types génériques dans de nombreux autres langages : Ada, Eiffel, Haskell, Caml...
Les generics existent pour faciliter la réutilisation du code, idée certes déjà fondamentale dans la programmation orientée Objet, mais qui est ici poussée un peu plus loin. Ils décrivent des classes et méthodes qui peuvent travailler de manière uniforme avec des valeurs de types différents. L'intérêt devient évident dès lors que l'on travaille avec un langage fortement typé : plus besoin de passer par le transtypage...
L'appellation du C++, "modèles", permet de mieux appréhender leurs fonctionnalités que "types génériques". En effet, ils permettent de définir des fonctions et méthodes qui s'adapteront aux types des paramètres qui leurs sont envoyé, ce qui permet de construire un véritable template, à la manière de ceux que l'on utilise pour créer un site Web : l'information, quelle qu'elle soit, s'intègre facilement au "moule" des generics, permettant de construire de nombreuses briques et de s'en servir, plutôt que de devoir travailler sur des routines précises pour des types précis.
Exemple
Les types génériques sont implémentés dans C# de la même manière que dans C++, afin de ne pas dérouter les programmeurs. Nous utilisons ici l'exemple classique de la classe Pile, qui permet de stocker des éléments. Cette classe possède deux méthodes : Push() permet d'introduire un élement dans la pile, Pop() de l'en sortir.

```
public class Pile<TypeDeLElement>
{
  private TypeDeLElement[] element;

  public void Push(TypeDeLElement data)
  {
    // code gérant l'introduction de l'élément.
  }

  public TypeDeLElement Pop()
  {
    // code gérant la sortie de l'élément.
  }
}

Pile<char> maPile = new Pile<char>();
maPile.Push("a");
char x = maPile.Pop();
```

La seule différence avec la déclaration habituelle de la classe est la présence des caractères < et >, qui permettent de définir le type que la pile doit avoir comme si c'était un paramètre de la classe. Lors de l'instanciation de la classe, il faut bien sûr déclarer le type des données utilisées.

On peut aussi utiliser plusieurs types pour une classe :

```
namespace Developper
{
    public class ClasseDemo< KeyType, ValueType >
    {
        private KeyType _obj;
        public ClasseDemo(KeyType obj1)
        {
           // constructeur
           _obj = obj1;
        }
        public ValueType Method(ValueType obj1, ValueType obj2)
        {
            // Code fonctionnel
        }
    }
}
```

Créez une table de hachage typée (ou trichez avec Dictionary, selon votre avancement). Utilisez les mécanismes d'introspection pour stocker les noms de méthodes, en clef, et la Pair (méthode - instance sur laquelle appeler la méthode), en valeur.

Demandez à l'utilisateur de saisir le nom de la méthode à appeler et l'exécuter.

Ajout de la description des méthodes

Les attributs permettent d'injecter de manière déclarative des métadonnées dans des types tels que les assemblies, les classes, les interfaces, les méthodes, etc. Nous allons expliquer ici comment définir des attributs personnalisés, comment les utiliser dans nos applications, et comment y recueillir grâce à la réflexion les propriétés des métadonnées au moment de l'exécution.

Créez une classe Export qui hérite de la classe Attribute.
Créez un membre public de type string appelé description.
Déclarez un attribut de type export pour la méthode hello en plaçant la ligne suivante juste au dessus de la déclaration de la méthode:
[Export(description=''method that displays hello world in the console'')]
Ajoutez à l'affichage des méthodes celui de leur description – Utilisation de la méthode `GetCustomAttributes` *de* `MethodInfo`