

.NET

C# - Delphi.NET

© 2006 Olivier Dahan – odahan@e-naxos.com – www.e-naxos.com

Les Weak References

Dans un environnement managé comme .NET la gestion des références à des instances de classe semble naturelle et sans poser de souci. Un objet est soit référencé, donc « vivant », soit n'est plus référencé, donc « mort » et éligible à sa destruction par le Garbage Collector (plus loin noté GC).

Si cela correspond le plus souvent au besoin, il est des cas où l'on voudrait conserver une référence sur un objet sans pour autant interdire son éventuelle libération. C'est le cas d'un cache d'objets par exemple : il référence des objets et peut les servir si besoin est, mais si la mémoire manque et que certains ont été détruits entre temps, cela n'est pas grave, ils seront recréés. Or, dans un environnement managé comme Java ou .NET, tant qu'une référence existe sur un objet ce dernier ne peut pas, par définition d'un environnement managé, être libéré. Dilemme... C'est là qu'interviennent les *références faibles*, ou **Weak References**. Nous allons ici voir comment ces références particulières sont mises en œuvre sous .NET à la fois sous C# et Delphi.NET.

Définition

Une référence faible est une référence à un objet qui bien qu'elle en autorise l'accès n'interdit pas la possible suppression de ce dernier par le Garbage Collector.

En clair, cela signifie qu'une référence faible, d'où sa « faiblesse », ne crée pas un lien fort avec l'instance référencée. C'est une référence qui, du point de vue du système de libération des instances n'existe pas, elle n'est pas prise en compte dans le graphe des chemins des objets « accessibles » par le GC.

C'est en fait comme cela que tout fonctionne en POO classique dans des environnements non managés comme Win32, rien n'interdit une variable de pointer un objet qui a déjà été libéré, mais tout accès par ce biais se soldera par une violation d'accès.

Sous environnement non managé il n'existe pas de solution simple pour éviter de telles situations, d'où l'engouement croissant pour les environnements managés comme .NET ou Java sans qu'aucun retour en arrière ne semble désormais possible, n'en déplaise à ceux qui restent attachés à Win32.

En effet, sous .NET une telle situation ne peut tout simplement pas arriver puisqu'on ne libère pas la mémoire explicitement, c'est le CLR qui s'en charge lorsqu'il n'y a plus de référence à l'objet.

Il ne faut d'ailleurs pas confondre mémoire et ressources externes. .NET protège la mémoire, pas les ressources externes. Pour cela les classes doivent implémenter `IDisposable`. Et si

un objet a été « disposé » il n'est pas « libéré » pour autant. On peut donc continuer à l'utiliser mais cela créera le plus souvent une erreur d'exécution puisque les ressources externes auront été libérées entre temps... Prenez une instance de `System.Drawing.Font`, appelez sa méthode `Dispose()`. Vous pourrez toujours accéder à l'objet en tant qu'entité, mais si vous tenter d'appeler sa méthode `ToString()` qui retourne le handle de l'objet fonte sous-jacent, une exception sera levée... Il existe une nuance importante entre mémoire et ressources externes, entre libération d'un objet et libération de ses ressources externes. C'est là l'une des difficultés du modèle objet de .NET qui pose souvent des problèmes aux débutants, et parfois même à des développeurs plus confirmés.

Le mécanisme

Le Garbage Collector du CLR libère la mémoire de tout objet qui ne peut plus être atteint. Un objet ne peut plus être atteint quand toutes les références qui le pointent deviennent non valides, par exemple en les forçant à `null`. Lorsqu'il détruit les objets qui se trouvent dans cette situation le GC appelle leur méthode `Finalize`, à condition qu'une telle méthode soit définie et que le GC en ait été informé (le mécanisme réel est plus complexe et sort du cadre de cet article).

Lorsqu'un objet peut être directement ou indirectement atteint il ne peut pas être supprimé par le GC. Une référence vers un objet qui peut être atteint est appelée une référence forte.

Une référence faible permet elle aussi de pointer un objet qui peut être atteint qu'on appelle la cible (*target* en anglais). Mais cette référence n'interfère pas avec le GC qui, si aucune référence forte n'existe sur l'objet, peut détruire ce dernier en ignorant les éventuelles références faibles (elles ne sont pas totalement ignorées puisque, nous allons le voir, la référence faible sera avertie de la destruction de l'objet).

Les références faibles se définissent par des instances de la classe **WeakReference**. Elle expose une propriété `Target` qui permet justement de réacquérir une référence forte sur la cible. A condition qu'elle existe encore... C'est pour cela que cette classe offre aussi un moyen de le savoir par le biais de sa propriété `IsAlive` (« est il encore vivant ? »).

Pour un système de cache, comme évoqué en introduction, cela est très intéressant puisqu'on peut libérer un objet (plus aucune référence valide ne le pointe) et malgré tout le récupérer dans de nombreux cas si le besoin s'en fait sentir. Cela est possible car entre le moment où un objet devient éligible pour sa destruction par le GC et le moment où il est réellement collecté et finalisé il peut se passer un temps non négligeable !

Le GC utilise trois « générations », trois conteneurs logiques. Les objets sont créés dans la génération 0, lorsqu'elle est pleine le GC supprime tous les objets inutiles et déplace ceux encore en utilisation dans la génération 1. Si celle-ci vient à être saturée le même processus se déclenche (nettoyage de la génération 1 et déplacement des objets encore valides dans la génération 2 qui représente tout le reste de la RAM disponible).

De fait, un objet qui a été utilisé un certain temps se voit pousser en génération 2, un endroit qui est rarement visité par le GC. Parfois même, s'il y a beaucoup de mémoire installée sur le PC ou si l'application n'est pas très gourmande, les objets de la génération 2, voire de la génération 1, ne seront jamais détruits jusqu'à la fermeture de l'application... A ce moment précis le CLR videra d'un seul coup tout l'espace réservé sans même finaliser les objets ni

appeler leur destructeur. C'est pourquoi sous .NET on ne programme généralement pas de destructeurs dans les classes : le mécanisme d'appel à cette méthode n'est pas déterministe.

Donc, durant toute la vie de l'application de nombreuses instances restent malgré tout en vie « quelque part » dans la RAM. Si une référence faible pointe l'un de ces objets il pourra donc être « récupéré » en réacquérant une référence forte sur lui par le biais de la propriété `Target` de l'objet `WeakReference`. Si l'objet en question réclame beaucoup de traitement pour sa création, il y a un énorme avantage à le récupérer s'il doit resservir au lieu d'avoir à le recréer, le tout sans pour autant engorger la mémoire puisque, si nécessaire, le GC l'aura totalement libéré, ce qu'on saura en interrogeant la propriété `IsAlive` de l'objet `WeakReference` qui aura alors la valeur `false`.

Si vous vous souvenez de ce que nous disions plus haut sur la nuance entre libération d'une instance et libération de ses ressources externes, vous comprenez que l'application ne doit utiliser des références faibles que sur des objets qui n'implémentent pas `IDisposable`. En effet, pour reprendre l'exemple d'une instance de la classe `Font`, si avant de mettre la référence à `null` votre application a appelé sa méthode `Dispose()`, récupérer plus tard l'instance grâce à une référence faible sera très dangereux : les ressources externes sont déjà libérées et toute utilisation de l'instance se soldera par une exception. Il est donc important de se limiter à des classes non *disposables*.

L'intérêt

Les références faibles ne servent pas qu'à mettre en œuvre des systèmes de cache, elles servent aussi lorsqu'on doit pointer des objets qui peuvent et doivent éventuellement être détruits. Rappelons-nous : si nous utilisons une simple référence sur un tel objet, il ne sera jamais détruit puisque justement nous le référençons... Les références faibles permettent d'échapper à ce mécanisme par défaut qui, parfois, devient une gêne plus qu'un avantage. Un exemple d'une telle situation : Supposons une liste de personnes. Cette liste pointe donc des instances de la classe **Personne**. Imaginons maintenant que l'application autorise la création de « groupes de travail », c'est-à-dire des listes de personnes. Si les listes définissant les groupes de travail pointent directement les instances de `Personne` et si une personne est supprimée de cette liste, les groupes de travail continueront de « voir » cette personne puisque l'instance étant référencée (référence forte) dans le groupe de travail elle ne sera pas détruite par sa simple suppression de la liste de base des personnes...

En fait, on souhaitera dans un tel cas que toute personne supprimée de la liste principale n'apparaisse plus dans les groupes de travail dans lesquels elle a pu être référencée. Cela peut se régler par une gestion d'événement : toute suppression de la liste des personnes entraînera le balayage de tous les groupes de travail pour supprimer la personne. Cette solution n'est pas toujours utilisable. Les références faibles deviennent alors une alternative intéressante, notamment parce qu'il n'y a pas besoin d'avoir prévu un lien entre la liste principale et les listes secondaires qui peuvent être ajoutées après coup dans la conception de l'application et parce que la suppression d'une personne n'impose pas une attente en raison du balayage de toutes les listes secondaires.

Mise en œuvre

Il est temps de voir comment implémenter les références faibles.

Finalement, vous allez le constater, c'est assez simple. Les explications qui précèdent permettent de comprendre pourquoi les références faibles sont utiles. Les utiliser réclame moins de mots...

La classe WeakReference

Cette classe appartient à l'espace de nom `System` du framework. Son constructeur prend en paramètre l'instance que l'on souhaite référencer (la cible).

Elle expose trois propriétés caractéristiques, les autres propriétés et méthodes étant celles héritées de la classe mère `System.Object` :

<code>IsAlive</code>	Indique si l'instance référencée est vivante ou non.
<code>Target</code>	Permet de réacquérir une référence forte sur la cible.
<code>TrackResurrection</code>	Pour dés/activer le pistage de résurrection de la cible.

Un mot sur cette dernière propriété : Lorsque l'on crée une référence faible on peut indiquer dans le constructeur, en plus de l'objet ciblé, un paramètre booléen qui fixera la valeur de `TrackResurrection`. Lorsque la valeur est « `false` » (par défaut) on parle de référence faible « courte », lorsque la valeur est « `true` » on parle de référence faible « longue ».

Si l'objet possède un finaliseur, c'est dans celui-ci qu'il sera possible d'indiquer ou non si l'instance doit rester en vie (être ressuscitée) ou pas, notamment par un appel à `GC.RegisterForFinalize(this)`. L'intérêt se trouve surtout dans les gestions de cache, car un objet « finalisable » survivra à au moins un cycle du GC en étant promu de la génération 0 à la génération 1. En retardant sa finalisation il sera poussé en génération 2 où il restera certainement un bon moment, améliorant ainsi grandement les chances de pouvoir le récupérer plus tard, donc rendant la gestion du cache encore plus efficace.

Le code

Le code qui suit est auto-documenté par sa simple exécution. S'agissant de projets console il vous suffit de créer une nouvelle application de ce type (que ce soit sous C# ou Delphi.Net) et de faire un copier / coller du code proposé. Lancez l'exécution (F5 sous VS2003/2005, F9 sous Borland Studio) et laissez-vous guider à l'exécution en jetant un œil sur le code...

Version C#

(application console, framework 1.1 pour compatibilité)

```
using System;
using System.Collections;

namespace od.article.WR
{
    class Class
    {
        public class Personne
        {
            private string _Nom;
            public string Nom
            {
                get { return _Nom; }
                set { _Nom = (value!=null) ? value.Trim() : string.Empty; }
            }

            public Personne(string nom)
            {
                this._Nom = nom.Trim();
            }
        }

        public static ArrayList Employés;

        private static string _line = "-----";
    }
}
```

```

        private static void Return()
        {
            Console.WriteLine("<return> pour continuer...");
            Console.ReadLine();
        }
        public static void ListeEmployés()
        {
            Console.WriteLine(_line);
            foreach ( Personne p in Employés)
                Console.WriteLine(p.Nom);
            Console.WriteLine(_line);
            Return();
        }

        [STAThread]
        static void Main(string[] args)
        {
            Employés = new ArrayList();
            Employés.Add(new Personne("Olivier"));
            Employés.Add(new Personne("Barabara"));
            Employés.Add(new Personne("Jacky"));
            Employés.Add(new Personne("Valérie"));

            Console.WriteLine("Liste originale");
            ListeEmployés();

            Personne p = (Personne)Employés[0]; // pointe "olivier"
            Employés.RemoveAt(0); // suppression de "olivier" dans la liste
            Console.WriteLine("p pointe : " + p.Nom);
            Console.WriteLine("L'élément 0 de la liste a été supprimé");
            Console.WriteLine();
            ListeEmployés();
            Console.WriteLine("Mais l'objet pointé par p existe toujours : "+p.Nom);
            Return();

            WeakReference wr = new WeakReference(Employés[0]); // pointe "barbara"
            Console.WriteLine(
                "wr est une référence faible sur: "+((Personne)wr.Target).Nom);
            Return();
            Console.WriteLine(
                "La cible de wr est vivante ? : "+wr.IsAlive.ToString());
            Return();
            Employés.RemoveAt(0); // suppression de "barbara"
            Console.WriteLine(
                "L'élément 0 ('barbara') a été supprimé. La liste devient :");
            ListeEmployés();
            Console.WriteLine(
                "La cible de wr est vivante ? : "+wr.IsAlive.ToString());
            Console.WriteLine(
                "On peut réacquérir la cible : "+((Personne)wr.Target).Nom );
            Return();
            Console.WriteLine("Mais si le GC passe par là...");
            GC.Collect(GC.MaxGeneration);
            Console.WriteLine(
                "La cible de wr est vivante ? : "+wr.IsAlive.ToString()); // false !
            Console.WriteLine(
                "La référence faible n'a pas interdit sa destruction totale.");
            Return();
        }
    }
}

```

Version Delphi.NET

(application console, framework 1.1)

```

program od.article.wr;

{$APPTYPE CONSOLE}

uses
    SysUtils,
    StrUtils,
    System.Collections;

type
    Personne = class
    private
        _Nom : string;
    public
        procedure setNom(const Value: string);
        property Nom:string read _Nom write setNom;
        constructor Create(nom:string);
    end;

{ Personne }

procedure Personne.setNom(const Value: string);
begin
    _nom := IfThen(Value<>nil, Value.Trim(), System.&String.Empty);
end;

constructor Personne.Create(nom: string);
begin
    inherited Create();
    _nom := nom.Trim();
end;

```

```

end;

{ fin Personne }

var Employés : ArrayList;
const _line = '-----';

procedure Return();
begin
    Console.WriteLine('<return> pour continuer...');
    Console.ReadLine();
end;

procedure ListeEmployés();
var p:Personne;
begin
    Console.WriteLine(_line);
    for p in Employés do Console.WriteLine(p.Nom);
    Console.WriteLine(_line);
    Console.WriteLine();
    Return();
end;

var p:Personne;
    wr:WeakReference;
begin
    Employés := ArrayList.Create();
    Employés.Add(Personne.Create('Olivier'));
    Employés.Add(Personne.Create('Barbara'));
    Employés.Add(Personne.Create('Jacky'));
    Employés.Add(Personne.Create('Valérie'));

    Console.WriteLine('Liste originale');
    ListeEmployés();

    p := Personne(Employés[0]); // pointe "olivier"
    Employés.RemoveAt(0); // suppression "olivier" dans liste

    Console.WriteLine('p pointe : '+p.Nom);
    Console.WriteLine('l'élément 0 de la liste a été supprimé');
    Console.WriteLine();

    ListeEmployés();

    Console.WriteLine('Mais l'objet pointé par p existe toujours : '+p.nom);

    Return();

    wr := WeakReference.Create(Employés[0]); // pointe "barbara"

    Console.WriteLine('wr est une référence faible sur : '+Personne(wr.Target).Nom);

    Return();

    Console.WriteLine('La cible de wr est vivante ? : '+wr.IsAlive.ToString());
    Employés.RemoveAt(0); // suppression de la liste de "barbara"

    Return();

    Console.WriteLine('l'élément 0 (barbara) a été supprimé, la liste devient : ');
    ListeEmployés();

    Console.WriteLine('La cible de wr est vivante ? : '+wr.IsAlive.ToString());
    Console.WriteLine('On peut récupérer la cible : '+Personne(wr.Target).Nom);

    Return();

    Console.WriteLine('Mais si le GC passe par là...');
    GC.Collect(GC.MaxGeneration);

    Console.WriteLine('La cible de wr est vivante ? : '+wr.IsAlive.ToString());
    Console.WriteLine('La référence faible n'a pas interdit la suppression de la cible.');
```

Conclusion

Cet article est un peu « austère », sans les habituels diagrammes UML qui font branché ni de jolies captures écrans qui enjolivent. Juste du texte et du code... Cela aura peut-être découragé certains lecteurs, alors à vous qui en êtes arrivé à cette conclusion, l'auteur à simplement envie de dire.. merci... Et bon développement !