Chapitre 7

Prérequis

☒Les chapitres précédents sont utiles. Le rappel sur la mémoire, le tas et la pile dans le chapitre Préparer efficacement la certification 70-483 peut également s'avérer utile.

Objectifs

☒Comprendre le fonctionnement du code managé. Comprendre le code non managé. Utiliser de façon avancée `Idisposable`. De manière générale, c'est le processus d'allocation et de désallocation de la mémoire sur le tas qu'il s'agit de bien comprendre ici.

A. Retour sur le ramasse-miettes

Le ramasse-miettes (*garbage collector* en anglais) constitue le dispositif clé de la gestion de la mémoire dans le cas d'un code managé. Or, le code C# est typiquement du code managé : en effet, ce code est géré à l'exécution, en l'occurrence par le CLR (*Common Language Runtime*).

En clair et en C#, l'humain qui développe délègue au CLR la gestion de la mémoire, mais également certains aspects de la sécurité et du typage. C'est donc l'exact opposé d'un langage comme le C++ pour lequel l'humain qui développe décide de quand et comment la mémoire est allouée et désallouée : le C++ est en effet un langage non managé.

Pour rappel, la mémoire est disponible dans deux endroits : la pile (*stack*) et le tas (*heap*). En C# :

- à la fin d'un *scope* donné (à la fin d'une méthode), l'utilisation de la mémoire sur la pile est nettoyée par le CLR. L'humain n'a rien à faire.
- le tas est, lui, géré par le ramasse-miettes qui est en charge de désallouer la mémoire allouée explicitement (grâce au mot-clé `new` par exemple) et qui n'est plus utilisée.

Il faut garder à l'esprit que le ramasse-miettes peut induire des considérations quant à la performance. Si le code doit répondre très rapidement et que dans le même temps le ramasse-miettes a énormément de mémoire à désallouer, on peut être amené à privilégier du code non managé dans des cas très précis.

B. Gérer le code « non managé »

Que peut bien être du code managé en C#, langage pourtant managé ? Eh bien typiquement des éléments de mémoire alloués par un autre programme et qu'il s'agit de désallouer dans votre code à vous. Dans cet exemple, il est évident que le ramasse-miettes n'a aucune connaissance de cette mémoire à désallouer : il s'agira donc de le faire à sa place de manière explicite.

Il est possible que la classe associée à la mémoire à désallouer fournisse une méthode de « finalisation », type de méthodes que nous allons expliquer dès maintenant.

1. La « finalisation » en C#

Fournir une telle méthode de finalisation (peu ou prou un destructeur dans d'autres langages) ne présume pas de son appel par le ramasse-miettes. On ne maîtrise pas le moment de l'appel de cette méthode. Par contre, on fournit l'implémentation : on maîtrise donc a minima la façon dont la désallocation va s'opérer.

Dans l'exemple suivant, `~MaClasse()` sera appelé à un moment donné par le ramasse-miettes. Quand exactement, on ne sait pas.

```
class MaClasse
{
    int[] notes;

    public MaClasse()
    {
        notes = new int[] { 12, 14, 8, 17 };
    }
```

```
        ~MaClasse()
        {
            // Ici le code appelé lors de la gestion par le ramasse-miettes.
        }
    }
```

Se pose alors la question du moment de l'appel du ramasse-miettes vis-à-vis d'instances données et donc d'allocations mémoire données. Peut-on forcer l'activation du ramasse-miettes ? La réponse est oui. Par exemple, une exception s'est déclenchée et il s'agit d'imposer la fermeture de flux, de fichiers, etc. Pour ce faire, on utilise l'objet `GC` (comme *garbage collector*).

```
GC.Collect();
GC.WaitForPendingFinalizers();
```

L'appel de la méthode `Collect` permet l'appel au ramasse-miettes ; `WaitForPendingFinalizers` permet la mise en attente de la conclusion effective de tous les finalizers (les méthodes de finalisation) appelés par le ramasse-miettes.

2. Rappels au sujet de l'interface IDisposable

Un objet utilisé peut implémenter l'interface `IDisposable` auquel cas il propose une implémentation de la méthode `Dispose`, à même de libérer les ressources allouées liées au dit objet. Dans l'exemple suivant, on crée un fichier physique grâce à `StreamWriter`, puis on libère les ressources allouées lors de cette création (car `StreamWriter` implémente `IDisposable` et fournit donc une méthode `Dispose`).

```
System.IO.StreamWriter sw = File.CreateText("MonFichier.txt");
sw.Write("On écrit dans le fichier grâce à notre flux sw");
sw.Dispose();
```

3. Créer un objet implémentant IDisposable

Pour cet exemple, nous allons manipuler un `IntPtr`, c'est-à-dire un pointeur de `int64` ou, exprimé autrement, le pointeur d'une adresse mémoire dont la mémoire pointée à cette adresse ne serait pas managée. Cela correspond peu ou prou à un `void*` en C++.

Le code suivant alloue de la mémoire non managée que la méthode `Dispose` est en capacité de désallouer.

```
class MonObjet : IDisposable
{
    private IntPtr mémoireNonManagéePointée;

    public MonObjet()
    {
        byte[] données = new byte[2];
        données[0] = 42;
        données[1] = 43;

        mémoireNonManagéePointée = Marshal.AllocHGlobal(sizeof(int));
        Marshal.Copy(données, 0, mémoireNonManagéePointée, données.Length);
    }
```

```
        public void Dispose()
        {
            Marshal.FreeHGlobal(mémoireNonManagéePointée);
            GC.SuppressFinalize(this);
        }
    }
```

☞*On observe que l'on réfère explicitement au ramasse-miettes dans la méthode* `Dispose` *pour qu'il soit « informé » de la désallocation de la mémoire associée à l'instance courante.*

☞*Bien se souvenir de la distinction entre méthode de finalisation (appelée uniquement par le ramasse-miettes) et méthode de type* `Dispose` *(qui est appelée explicitement dans le code).*

Ci-dessous le code complet de l'exemple précédent :

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

namespace ConsoleMonObjet
{
    class MonObjet : IDisposable
    {
        private IntPtr mémoireNonManagéePointée;

        public MonObjet()
        {
            byte[] données = new byte[2];
            données[0] = 42;
            données[1] = 43;

            Console.WriteLine("Allocation mémoire.");
            mémoireNonManagéePointée = Marshal.AllocHGlobal(sizeof(int));

            Console.WriteLine("Copie des données dans la mémoire allouée.");
            Marshal.Copy(données, 0, mémoireNonManagéePointée, données.Length);
        }

        public void Dispose()
        {
            Console.WriteLine("Libération de la mémoire allouée");
            Marshal.FreeHGlobal(mémoireNonManagéePointée);
            GC.SuppressFinalize(this);
        }
    }

    class Programme
    {
        static void Main(string[] args)
        {
```

```
            Console.WriteLine("Instanciation de l'objet");
            MonObjet mo = new MonObjet();
            Console.WriteLine("Appel explicite de Dispose()");
            mo.Dispose();
        }
    }
}
```

On obtient la sortie suivante dans la console :

```
Instanciation de l'objet
Allocation mémoire.
Copie des données dans la mémoire allouée.
Appel explicite de Dispose()
Libération de la mémoire allouée
```

C. Pointeurs faibles

Un pointeur faible (ou référence faible, *weak reference* en anglais) permet de maintenir une référence forte sur une donnée. En clair, sous réserve que le ramasse-miettes ne soit pas encore intervenu, on peut grâce à une référence faible accéder à une donnée qui aurait été désallouée par le ramasse-miettes avec une référence classique.

.Net propose une classe pour définir un pointeur faible : `WeakReference`, définie dans l'espace de noms `System`.

```
public class WeakReference : ISerializable
{
  public WeakReference(object target);
  public WeakReference(object target, bool trackResurrection);
  protected WeakReference(SerializationInfo info, StreamingContext context);

  ~WeakReference();

  public virtual bool IsAlive { get; }
  public virtual bool TrackResurrection { get; }
  public virtual object Target { get; set; }

  public virtual void GetObjectData(SerializationInfo info,
StreamingContext context);
}
```

Dans l'exemple suivant, on définit une référence faible avec la classe `WeakReference` autour d'un `StringBuilder` (étudié en détail dans le chapitre suivant). On sollicite explicitement le ramasse-miettes, mais comme le montre l'écriture dans la console, la valeur est toujours disponible.