Destructeurs d'objet et Finaliseurs sous .NET

MCours.com

1. Public concerné

Débutant	Avancé	Confirmé

Testé sous le framework .NET 1.1 et Delphi 2005.
Version 1.0

Cet article contient des liens nommés **Local**. Ils pointent sur des URLs locales de la documentation de Microsoft. Leur utilisation nécessitera l'installation du SDK .NET v1.1 en Français sur le poste.

Je tiens à remercier **neo.51**, **Nono40** et **Piotrek** pour leur remarques pertinentes.
Un grand merci à Olivier Lance **(Bestiol)** pour les corrections orthographiques.

1-1. Les sources

Les fichiers sources des différents exemples :
FTP.
HTTP.

L'emplacement du répertoire, contenant le fichier projet Delphi correspondant à un exemple, est indiqué en commentaire dans le code :

```
//Répertoire : Destructeurs-
finaliseurs\Managed Code\Finalizer
```

Les fichiers sources proposés dans l'article d'origine contiennent le code C# et C++, le code Delphi était destiné à une version *Bêta* de Delphi 8.

2. Destructeurs d'objet et Finaliseurs

2-1. Introduction

Dés que nous quittons les modèles de programmation traditionnels sous Windows pour entrer dans l'environnement de programmation de .NET nous devons ajuster notre manière d'écrire nos classes. En particulier nous devons repenser notre approche des destructeurs, car leur rôle et leur comportement diffèrent sous le Common Language Runtime (CLR) de .NET.

Cet article explique le nouveau rôle des destructeurs, comment ils sont considérés du point de vue de la CLR, ce qui vous devez faire et ne pas faire dans un destructeur. Ceci concerne l'écriture de composants, tout autant que l'écriture d'application. Vous verrez que l'utilisation du code de finalisation d'objets (tels que des destructeurs) est une situation beaucoup plus rare sous .NET que sous Win32 ou Linux.

L'article d'origine prend en considération ces problématiques sous le langage C#.

2-2. La destruction déterministe

Quand vous écrivez une classe en Delphi vous vous basez sur le fait que c'est le programmeur qui construit une instance de votre classe, qui est responsable de sa destruction une fois l'objet devenu inutile. Détruire l'instance implique d'appeler le destructeur de l'objet qui procède à la libération de toutes les ressources spécifiques utilisées par l'objet, tel que des blocs de mémoire, d'autres objets ou des ressources de l'OS. Il libère également la mémoire occupée par les données de l'instance de l'objet.

Notez qu'un destructeur dans un langage de programmation managé s'occupe de deux tâches :

1. libérer les ressources employées par l'objet
2. libérer la mémoire occupée par les données de l'instance de l'objet

Si vous avez une hiérarchie de classes, quelques-unes ou toutes peuvent définir des destructeurs spécifiques pour libérer les ressources dont chaque classe individuelle se sert. Quand vous détruisez un objet, son destructeur libère ses ressources et enchaîne de nouveau l'appel du destructeur de la classe parente et ainsi de suite jusqu'à la classe de base. Quand tous les destructeurs ont été appelés, les données en mémoire de l'instance sont libérées.
Comme la plupart des ressources de Windows utilisent des "Handles", étudions la gestion d'un de ces Handle dans le cadre d'une classe simple. En réalité nous pourrions hériter d'une variété de classes, mais pour simplifier nous emploierons une seule classe.

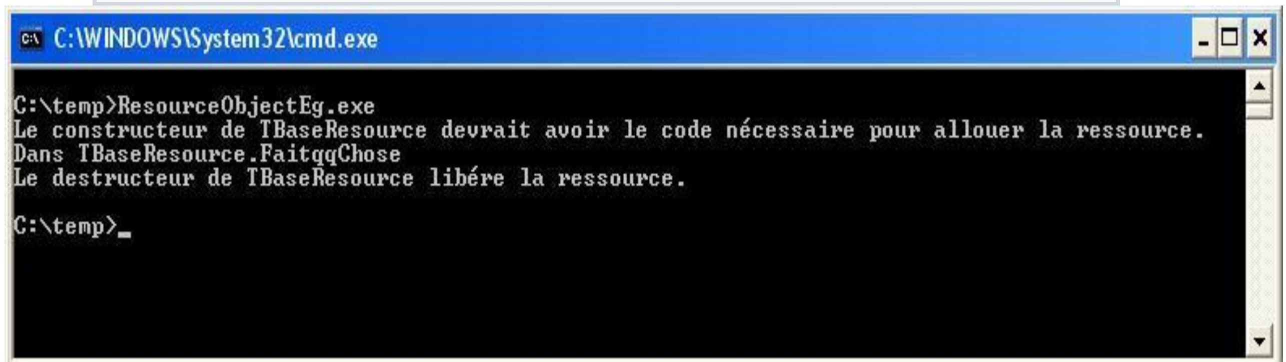
2-3. Destructeur sous Delphi Win32

Sous Win32, Delphi supporte uniquement des objets basés sur une allocation dans le tas (Heap) accessibles par des références d'objet. Les références d'objet sont des pointeurs mais ne nécessitent pas la syntaxe standard de pointeur (^).
Les objets sous Delphi sont construits en appelant le constructeur de la classe concernée, par convention nommé **Create**, et sont détruits en appelant le destructeur de la classe, nommé**Destroy**.
Les destructeurs sous Delphi sont polymorphes et par convention ne sont pas appelés directement mais indirectement par un appel à la méthode **Free** :

```
program ResourceObjectEg;
//Répertoire : Destructeurs-
finaliseurs\Unmanaged Code
{$APPTYPE CONSOLE}
```

```
uses
  Windows;

type
  TBaseResource = class(TObject)
  private
    // La ressource est un handle
    handle: THandle;
  public
    constructor Create;
    destructor Destroy; override;
    procedure FaitqqChose;
  end;

constructor TBaseResource.Create;
begin
  inherited Create;
  handle := INVALID_HANDLE_VALUE;
  WriteLn('Le constructeur de TBaseResource
devrait avoir le code nécessaire pour allouer
la ressource.');
end;

destructor TBaseResource.Destroy;
begin
  WriteLn('Le destructeur de TBaseResource
libère la ressource.');
  if handle <> INVALID_HANDLE_VALUE then
    CloseHandle(handle);
  inherited;
end;

procedure TBaseResource.FaitqqChose;
begin
  WriteLn('Dans TBaseResource.FaitqqChose.');
end;

var
  BR: TBaseResource;
begin
  BR := TBaseResource.Create;
  try
    BR.FaitqqChose;
  finally
```

```
    BR.Free;
  end
end.
```



Résultat

2-4. Commentaires sur les destructeurs

Ce processus de destruction explicite d'objet est désigné sous le terme de destruction déterministe et est très répandu dans les langages de programmation orientés objet. Les programmeurs de ces langages le considèrent comme la norme. Malheureusement la destruction déterministe cause un nombre important de bogues difficiles à pister pendant et après le cycle de développement des applications, simplement parce qu'il est de la responsabilité du programmeur de détruire un objet et de le détruire au moment opportun.

L'erreur étant humaine, il est courant que les objets ne soient pas tous détruits, créant ainsi des fuites mémoire. Il est courant qu'un objet soit détruit puis de nouveau référencé un peu plus loin dans le code, provoquant ainsi des corruptions de données ou des violations d'accès. Il peut être difficile de tracer ces deux types de problèmes, à la différence de beaucoup de problèmes de logique. Heureusement pour nous les développeurs, ces maux de tête cessent sous .NET qui nous dispense de ces exigences pour la destruction déterministe.

2-5. Finalisation non déterministe

Le CLR emploie le procédé appelé *garbage collection* pour éviter les bogues décrits ci-dessus. Le programmeur n'a plus la responsabilité de détruire les objets devenus inutiles ; en fait il n'est pas possible qu'un programmeur détruise un objet parce que la notion conventionnelle du destructeur a disparu. Les objets sous .NET sont assignés sur un tas managé. Quand des objets ne sont plus référencés par aucune variable dans une application (objets alors inaccessibles par n'importe quel code), ils ont clairement besoin d'être libérés. Cependant le programmeur n'a rien à faire pour s'assurer que ceci se produira. Au lieu d'être de la responsabilité du programmeur, ceci est désormais sous le contrôle du Garbage Collector (*ramasse-miettes*).

*Le ramasse-miettes étant la traduction française de **Garbage Collector**, j'utiliserai le terme de balayage pour désigner le processus de libération de la mémoire (**Garbage Collection**) et quelques fois le terme de miettes pour désigner les objets devenus inutiles.*

La prochaine fois qu'une opération de nettoyage du tas contrôlé s'exécute, tous les objets inaccessibles seront identifiés, leur mémoire sera récupérée et le tas managé compacté. Savoir quand le ramasse-miettes effectuera réellement cette opération est difficile à prévoir en réalité, toutefois l'algorithme qu'il emploie est bien documenté. Vous pouvez également forcer l'exécution d'une opération de nettoyage si besoin est, bien que de telles situations soient rares.

Il s'avère ici que sans intervention d'aucune sorte du programmeur, les données d'instance occupées par n'importe quel objet .NET seront automatiquement libérées après qu'il aura fini de les utiliser. Ceci signifie qu'une part significative du travail du destructeur est évitée.

Il y a également d'autres tâches importantes qu'un destructeur doit prendre en compte. Un destructeur devrait libérer toutes les ressources additionnelles utilisées par l'objet :

- Si l'objet possède des références sur d'autres objets, le destructeur traditionnel devrait aussi les détruire. Ce n'est plus nécessaire dans .NET car le ramasse-miettes récupérera leur mémoire d'instance à un certain moment.
- Les ressources non managées, telles que les connexions de base de données et les handles de fichier, etc. doivent être libérées mais le ramasse-miettes ne prend pas en charge la récupération de l'espace mémoire de ce type de données.

Pour être clair, il n'est pas commun que les classes de .NET utilisent directement des ressources non managées (unmanaged). Il est préférable d'employer d'autres objets de .NET qui prennent en charge la libération des ressources non managées encapsulées. Par exemple, les fichiers sont habituellement représentés par des instances de la classe **System.IO.FileStream** et les connexions de base de données pourraient être représentées soit par des instances de **System.Data.SqlClient.SqlConnection** ou de **System.Data.OleDb.OleDbConnection**.

Il est de la responsabilité des développeurs dans ce cas de prendre en charge directement la gestion de ce type de ressource non managée. De temps à autres, cependant, vous pourriez être dans une situation où vous devez savoir comment vous débarrasser correctement des ressources non managées. Heureusement, tous les objets de .NET offrent une occasion de s'assurer que des ressources non managées sont correctement libérées par leur méthode **Finalize**.

Les programmeurs d'applications ont leurs propres problèmes qui doivent être abordés, puisqu'ils travaillent souvent avec des objets managés qui contrôlent des ressources non managées (telles que des fichiers ou des connexions de base de données) et ils peuvent souhaiter libérer ces ressources avant que le ramasse-miettes ne le fasse. Nous aborderons cette question sous peu.

2-6. Finaliseurs

La classe de base dans l'environnement .NET **System.Object**, propose la méthode virtuelle et protégée **Finalize**. Son implémentation ne fait rien, mais elle peut être surchargée par n'importe quelle classe qui nécessite le nettoyage de ressources non managées. Une classe qui surcharge **Finalize** est parfois décrite comme ayant un finaliseur (*finalizer*) ou comme étant un objet finalisable (*finalizable object*).

Quand le ramasse-miettes détermine qu'un objet est inaccessible, il vérifie s'il possède un finaliseur et dans ce cas l'appelle avant de récupérer la mémoire de l'objet (son implémentation réelle est abordée plus loin). Ceci permet maintenant à un objet managé de .NET d'offrir le même comportement de nettoyage que celui atteint avec un destructeur d'objet non managé. Le ramasse-miettes appelle le finaliseur afin de libérer les ressources non managées spécifiques puis procède à la récupération de la mémoire des données de l'instance.

Rappelez-vous que les conditions nécessitant l'utilisation d'un finaliseur sont rares et s'appliquent seulement quand vous avez des ressources non managées qui doivent être libérées.

2-7. Quelques questions au sujet d'un finaliseur

Le problème principal auquel nous devons faire face avec un finaliseur est que nous n'avons toujours pas de contrôle sur le moment où le ramasse-miettes se déclenchera et libérera notre objet (appeler le finaliseur s'il est présent et récupérer également la mémoire de l'objet). Si la ressource gérée par l'objet est libérée rapidement (comme une connexion de base de données non managée), la simple surcharge de la méthode **Finalize** ne nous aidera pas.

Cette problématique est décrite par le terme de *finalisation non déterministe*. La finalisation de l'objet se produira, mais pas quand le programmeur le souhaite, au lieu de cela il se produira quand le ramasse-miettes s'en chargera.

Puisque la méthode **Finalize** est protégée, elle ne peut pas être appelée directement par un client de l'objet pour résoudre ce problème. Cependant vous pourriez appliquer une méthode publique qui appelle **Finalize** et l'utiliser pour permettre aux clients de libérer les ressources de l'objet à un instant donné. Si vous deviez faire ceci, vous devriez alors dire au ramasse-miettes qu'il n'a pas besoin d'appeler le finaliseur lorsqu'il s'occupera de l'objet.

Cette approche n'est pas recommandée principalement pour deux raisons. Premièrement, il existe un mécanisme formel conçu pour aider la finalisation déterministe, que nous regarderons plus tard : *le pattern Dispose*. Deuxièmement, en C# il n'est pas possible d'appeler explicitement le finaliseur de n'importe quelle autre méthode, comme nous le verrons dans la prochaine section (***cf. article d'origine***). Naturellement nous pourrions résoudre ce problème en appliquant une méthode publique qui libèrerait les ressources non managées, et serait appelée depuis le finaliseur avec les contrôles appropriés, mais là encore l'approche formelle est préférée.

Un autre problème est que vous ne pouvez pas garantir l'ordre dans lequel les finaliseurs des objets s'exécuteront, même lorsqu'un objet contient une référence sur un autre objet et que tous les deux possèdent des finaliseurs. Ceci signifie que les finaliseurs ne devraient jamais accéder à d'autres objets finalisables (les objets sans finaliseur sont préférables) ; ils devraient généralement être limités à la libération de la ressource non managé.

2-8. Finaliseurs sous C#

Voir **l'article d'origine**.

2-9. Finaliseurs sous Delphi .NET

Avec Delphi pour .NET vous êtes libre de surcharger **Finalize** dans vos classes et si vous en sentiez le besoin, de déclarer une méthode publique pour exposer la méthode **Finalize** aux clients de vos objets (là encore, vous devriez typiquement employer le *pattern Dispose* au lieu de procéder de la sorte).

Voici comment pourrait être écrit une classe nécessitant des ressources non managées en utilisant un finaliseur :

```
program ResourceObjectEg;
//Répertoire : Destructeur-finaliseurs\Managed
Code\Finalizer

{$APPTYPE CONSOLE}

{$WARN UNIT_PLATFORM OFF}
uses Borland.Vcl.Windows; // Provoque un
avertissement concernant la plate-forme.

type
  TBaseResource = class
  private
    // La ressource est un handle
    handle: IntPtr;
  Strict protected
    procedure Finalize; override;
  public
    constructor Create;
    procedure FaitqqChose;
  end;

constructor TBaseResource.Create;
begin
  inherited Create;
  handle := IntPtr.Zero;
  WriteLn('Le constructeur de TBaseResource
devrait avoir le code nécessaire pour allouer
la ressource.');
end;
```

```
procedure TBaseResource.Finalize;
begin
  try
    WriteLn('Le finaliseur de TBaseResource
libère la ressource.');
    if handle <> IntPtr.Zero then
      // API Win32
      CloseHandle(handle.ToInt32);
  finally
    inherited
  end
end;

procedure TBaseResource.FaitqqChose;
begin
  WriteLn('Dans TBaseResource.FaitqqChose');
end;

var
  BR: TBaseResource;
begin
  BR := TBaseResource.Create;
  BR.FaitqqChose;
  WriteLn('Aucune libération explicite dans
Main.');
end.
```

```
C:\WINDOWS\System32\cmd.exe

C:\temp>ResourceObjectEg.exe
Le constructeur de TBaseResource devrait avoir le code nécessaire pour allouer la ressource.
Dans TBaseResource.FaitqqChose
Aucune libération explicite dans Main.
Le finaliseur de TBaseResource libére la ressource.

C:\temp>
```

Résultat

Notez qu'il n'y a aucun appel à **BR.Free** comme dans la version non managé, vous pourriez toutefois en placer un mais cela n'aurait aucun effet sur le code. Nous reviendrons plus tard sur la méthode **Free**.

2-10. Détails sur le Garbage Collector (ramasse-miettes)

En vérité, le fonctionnement du Garbage collector (ramasse-miettes) est plus compliqué que ce qui en a été dit jusqu'à maintenant. Ce chapitre explore en profondeur le mode de fonctionnement du ramasse-miettes afin de clarifier certains problèmes.

2-10-1. Algorithme de génération

Pour améliorer l'efficacité d'exécution le ramasse-miettes utilise des générations. Les générations sont des divisions logiques du tas contrôlé (*managed heap*). Le CLR emploie trois générations : la génération 0, la génération 1 et la génération 2. Les nouveaux objets sont toujours assignés dans la génération 0 du tas.

Quand un objet est alloué et que la génération 0 est pleine, il ne peut donc pas être créé dans cette génération, dans ce cas le ramasse-miettes commencera un balayage de la génération 0 qui correspond à une recherche des objets inaccessibles dans la génération 0 puis récupérera leur mémoire. Tous les objets accessibles sont alors promus dans la zone de génération 1 du tas, laissant de ce fait la génération 0 vide.

Tous les objets déplacés en mémoire par cette promotion de générations voient toutes leurs références dans l'application mises à jour pour refléter leur nouvelle adresse. Ceci signifie que vous ne pouvez pas supposer qu'un objet demeurera pendant sa durée de vie à l'adresse où il a été alloué à l'origine ; il peut être déplacé par le ramasse-miettes. Si besoin vous pouvez figer un objet à sa place actuelle ainsi il ne sera pas déplacé (employer le mot-clé fixed en C# ou la structure ***System.Runtime.InteropServices.GCHandle*** *et les méthodes****Alloc*** *et* ***Free****).*

Pendant une opération de nettoyage de la mémoire, tandis que des objets de la génération 0 sont promus vers la génération 1 il se peut que la génération 1 soit pleine pour les recevoir tous ou en partie. Dans ce cas le ramasse-miettes balayera la génération 1, récupérant ainsi la mémoire des objets inaccessibles et promouvant les objets accessibles vers la génération 2.

À un certain point, tout en promouvant les objets de la génération 1 vers la génération 2, il se peut que la génération 2 soit remplie. Dans ce cas, le ramasse-miettes balayera la génération 2 et récupérera la mémoire des objets inaccessibles regagnant un peu d'espace. Les objets accessibles dans la génération 2 demeurent dans la génération 2.

Ainsi la génération 0 contient les nouveaux objets qui n'ont pas été examinés par le ramasse-miettes, la génération 1 contient les objets qui ont été examinés une fois par le ramasse-miettes (et étaient encore accessibles à ce moment) et la génération 2 contient les objets qui ont été examinés au moins deux fois (et étaient encore accessibles).

L'idée derrière ce type d'algorithme de nettoyage de la mémoire (garbage collection) implique un certain nombre de suppositions :

- Les nouveaux objets auront une durée de vie courte. Ils sont alloués dans la génération 0 et la génération 0 est ce que le ramasse-miettes examine par défaut.
- Les objets plus anciens auront une durée de vie plus longue. Ils sont promus vers la génération 1 ou la génération 2, qui sont des zones du tas contrôlé moins fréquemment balayées.
- Il est plus efficace d'opérer sur une partie du tas, plutôt que de tout contrôler, c'est pourquoi le ramasse-miettes balaye seulement la génération 0 par défaut. Les essais de performance effectués par Microsoft indiquent qu'une opération de nettoyage dure entre 0 et 10 millisecondes pour la génération 0. Une opération de nettoyage dans la génération 1 dure typiquement entre 10 et 30 millisecondes.

MCours.com

Il a été mentionné précédemment que vous pouviez forcer, si besoin est, le ramasse-miettes à opérer un nettoyage du tas contrôlé. Ceci est réalisé en appelant **System.GC.Collect**, mais vous devez être conscient qu'**il n'est pas recommandé d'effectuer ce type d'appel**. Premièrement, la principale raison des personnes appelant manuellement un balayage par le ramasse-miettes est de réduire les lenteurs périodiques dans l'application quand celui-ci se produit naturellement. Ils appelleront le ramasse-miettes pendant des opérations d'UI (ou d'autre traitement long de l'application) ainsi la surcharge n'est pas apparente. Cependant les informations de chronométrage fournies ci-dessus démontrent que l'opération de nettoyage n'est pas typiquement une opération longue.

La plupart des informations sur la génération contrôlée de tas indiquent leurs tailles comme suit :

- La génération 0 commence avec un seuil autour 256 Kb
- La génération 1 commence avec un seuil autour de 2 Mo
- La génération 2 commence avec un seuil autour de 10 Mo

Cependant des ingénieurs CLR de Microsoft suggèrent que les seuils de la génération 0 et de la génération 1 commencent à des niveaux différents. Le seuil de la génération 0 se réfère à la taille du cache L2 du processeur (également appelée le cache de niveau 2 ou cache secondaire). Le seuil minimum initial pour la génération 1 se situe autour de 300 Kb, tandis que la taille maximum peut être la moitié de la taille d'un segment, qui pour un simple poste de travail mono-processeur habituel s'élèvera à 8 Mo. Le plan étant que la plupart des allocations de la génération 0 (i.e objets contrôlés) vivront et mourront entièrement dans le cache très rapide L2 du CPU.

Le ramasse-miettes tout en fonctionnant, surveille la façon dont l'application alloue la mémoire (via la construction d'objet). Au besoin il modifiera les seuils de chacune des générations de tas contrôlé. La deuxième raison de ne pas appeler manuellement le ramasse-miettes est que ceci annulera son analyse statistique du

programme, qu'il emploie pour décider des réglages au plus fin de ces seuils.

Voici un exemple visualisant le fonctionnement des générations :

```
program GCCollect;
// Destructeur-finaliseurs\Managed Code\GC-
Collect

{$APPTYPE CONSOLE}

Const CR_LF=#13#10;

type
  TBaseResource = class
  Strict Protected
    procedure Finalize; override;
  end;

procedure TBaseResource.Finalize;
begin
  try
    WriteLn('Le finaliseur de TBaseResource
libère la ressource.');
    Writeln('L''objet se trouve dans la
génération numéro
'+GC.GetGeneration(Self).ToString);
  finally
    inherited
  end
end;

var
  BR: TBaseResource;
begin
  Writeln('Nombre de génération maximum
'+GC.MaxGeneration.ToString+CR_LF);
   // Création d'un objet dans le Heap
  BR := TBaseResource.Create;
   //Affiche la génération dans laquelle il a
été crée
  Writeln('L''objet BR se trouve dans la
génération numéro
'+GC.GetGeneration(BR).ToString);
```

```
  GC.Collect;

  Writeln('L''objet BR se trouve dans la
génération numéro
'+GC.GetGeneration(BR).ToString);
  GC.Collect;

  Writeln('L''objet BR se trouve dans la
génération numéro
'+GC.GetGeneration(BR).ToString);
  GC.Collect;

  BR:=Nil; // Détruit la référence forte de
l'objet;

  // La méthode GC.GetGeneration(BR) ne peut
plus être appelé sinon elle déclenche
l'exception System.ArgumentNullException
  // Writeln('Nettoyage de la génération
numéro '+GC.GetGeneration(BR).ToString);

  Writeln(CR_LF+'Nettoyage de la génération
numéro 0');
  GC.Collect(0);
  GC.WaitForPendingFinalizers; // Le
finaliseur n'est pas appelé

  Writeln('Nettoyage de la génération numéro
1');
  GC.Collect(1);
  GC.WaitForPendingFinalizers; // Le
finaliseur n'est pas appelé

  Writeln('Nettoyage de la génération numéro
2'+CR_LF);
  GC.Collect(2);
  GC.WaitForPendingFinalizers; // Le
finaliseur est appelé

  Readln;
end.
```

```
c:\temp>GCCollect.exe
Nombre de génération maximum 2

L'objet BR se trouve dans la génération numéro 0
L'objet BR se trouve dans la génération numéro 1
L'objet BR se trouve dans la génération numéro 2
Le finaliseur de TBaseResource libère la ressource.

Nettoyage de la génération numéro 0L'objet trouve dans la génération numéro 2

Nettoyage de la génération numéro 1
Nettoyage de la génération numéro 2
```

Résultat

A voir aussi :
Gestion automatique de la mémoire. **Local.**
La méthode **GC.WaitForPendingFinalizers** [C#]. **Local.**

Quelques arguments supplémentaires concernant l'utilisation de GC.Collect :
Two things to avoid for better memory usage
When to call GC.Collect
The perils of GC.Collect

2-10-2. D'autres questions sur le finaliseur

Nous avons déjà abordé le sujet, toutefois il y a quelques détails que nous devons connaître en ce qui concerne l'exécution des finaliseurs afin d'apprécier pleinement la situation. Quand un objet possédant un finaliseur est créé, le CLR ajoute une référence sur cet objet dans une liste interne appelée la liste de finalisation (**finalization list**). Ceci facilite pour le ramasse-miettes la reconnaissance des objets qui nécessitent une finalisation avant de récupérer leur mémoire. Notez que ceci impacte peu la construction de tous les objets qui ont des finaliseurs.

Quand le ramasse-miettes balaye le tas managé et trouve les objets qui sont inaccessibles par du code de programme, il vérifie si l'un d'entre eux apparaît dans la liste de finalisation. Ceux qui

en ont besoin appellent leurs finaliseurs et ne peuvent donc pas être prêts pour récupérer leur mémoire au même moment. Une référence à ces objets est ajoutée à une autre liste interne, appelée la file d'attente d'accessibilité (**freachable queue**) et leur référence dans la liste de finalisation est supprimée. Ceci nous indique que les objets finalisables ralentissent le ramasse-miettes, puisque chaque balayage lors d'une opération de nettoyage doit être accompagné de recherches dans la liste de finalisation (un contrôle est effectué pour chaque objet inaccessible pour voir s'il est dans la liste de finalisation).

L'idée de la file d'attente d'accessibilité est la suivante : étant donné que les objets contenus dans cette liste doivent exécuter une de leur propre méthode (le finaliseur), il doivent donc toujours être considérés comme accessibles. Pour cette raison ces objets sont favorisés jusqu'à la prochaine génération du tas, la transition des objets considérés comme inaccessibles en objet accessibles (quoique temporairement), permet de ce fait au ramasse-miettes de récupérer n'importe quelle mémoire d'objets qui étaient inaccessibles et n'ayant pas de finaliseurs.

Que se passe-t-il pour les objets qui ont survécu au ramasse-miettes ? Eh bien, il y a un thread dédié de haute priorité (**le thread finaliseur**) contrôlé par le CLR qui surveille de près la file d'attente d'accessibilité. Quand la file d'attente est vide le thread finaliseur se met en sommeil, mais lorsque des objets sont ajoutés à la file d'attente il est réveillé et commence à appeler séquentiellement leurs finaliseurs.

A partir de là nous apprenons que des finaliseurs ne sont pas appelés de notre thread d'application. Ceci implique qu'il est important de s'assurer que votre finaliseur fonctionne aussi rapidement que possible et ne fait aucun blocage (attente d'un autre thread ou d'une autre ressource pour devenir disponible), comme indiqué plus loin.

Dès qu'un finaliseur d'objet a été appelé, l'objet est supprimé de la file d'attente d'accessibilité et est dès lors vraiment considéré comme une 'miette' attendant d'être 'ramassée'. Toutefois ceci ne se produira que la prochaine fois où le ramasse-miettes balaiera

le tas occupé par l'objet, qui ne sera généralement pas dans le tas de génération 0 (puisqu'il a été promu lors du déplacement de la liste de finalisation vers la file d'attente d'accessibilité), et ceci se produira certainement plus tard lors de la prochaine opération de nettoyage qui examinera juste le tas de génération 0.

Ceci nous indique que les objets avec des finaliseurs ont une durée de vie beaucoup plus longue que ceux sans, et qu'ils nécessitent au moins deux opérations de nettoyage pour libérer leur mémoire. Une conséquence importante est que tous les autres objets référencés par l'objet finalisable (et tous les objets qui référencent ces objets, et ainsi de suite) seront également gardés beaucoup plus longtemps que vous l'aviez prévu, puisqu'ils demeureront accessibles jusqu'à ce que l'objet finalisable soit finalisé.

En fait un objet finalisable peut également être ressuscité pendant l'exécution de son finaliseur, prolongeant un peu plus sa durée de vie. Un appel à **System.GC.ReRegisterForFinalize** ajoute l'objet passé en paramètre à la liste de finalisation (il y a peu de circonstances où cela est bénéfique), s'assurant ainsi et après qu'il a été considéré comme inaccessible que son finaliseur sera appelé de nouveau.

Tandis que les finaliseurs sont appelés séquentiellement pour les objets présents dans la file d'attente d'accessibilité, vous ne pouvez pas prévoir dans quel ordre les objets sont passés dans la file d'attente (cela dépend de l'ordre dans lequel le ramasse-miettes découvre qu'ils sont inaccessibles, ceci ne pouvant être déterminé). Ceci signifie que vous ne pouvez pas prévoir dans quel ordre les finaliseurs sont appelés. Même lorsqu'un objet, déclarant un finaliseur, contient une référence à un autre objet ayant lui aussi un finaliseur, les deux finaliseurs pourraient être appelés dans un ordre ou dans l'autre.

Ceci signifie qu'un finaliseur ne doit se rapporter à aucun autre objet possédant un finaliseur, en présumant qu'un finaliseur a ou n'a pas été appelé. En général les finaliseurs devraient simplement libérer les ressources de l'objet et ne faire rien d'autre.

Si une exception non-gérée se produit dans un finaliseur, le CLR l'ignore et met fin à cette méthode finalize, l'enlève de la file d'attente d'accessibilité et passe à la prochaine entrée.

Des cas plus sérieux peuvent cependant se produire si votre finaliseur ne se termine pas, par exemple s'il se bloque en attendant une condition qui ne se produira jamais. Dans ce cas-ci le thread finaliseur sera suspendu provoquant ainsi l'arrêt de l'opération de nettoyage des objets finalisables. Vous devez être attentif à cette situation en vous efforçant d'écrire dans les finaliseurs le code le plus simple possible pour libérer vos ressources non managées.

Considérez également ce qui se produit pendant l'arrêt d'une application. Quand le programme se termine, le ramasse-miettes essayera d'appeler les finaliseurs de tous les objets finalisables, mais avec certaines limitations :

- pendant l'arrêt les objets finalisables ne sont pas promus vers des générations de tas plus élevées.
- N'importe quel finaliseur individuel aura un maximum de 2 secondes pour s'exécuter ; Si cela prend plus de temps il sera tué (killed).
- Il y a un maximum de 40 secondes pour que tous les finaliseurs soient exécutés ; Si des finaliseurs s'exécutent toujours ou sont en attente à cet instant, le processus entier est brusquement tué.

Ces délais d'attente pourraient être modifiés dans le futur.

L'invocation des finaliseurs dépend du ramasse-miettes et est ainsi non-déterministe, ce qui peut être inadéquat pour certains types de ressource, par exemple des connexions de base de données non managées. La prochaine section aborde une démarche permettant la finalisation déterministe.

Il est toutefois possible d'appeler manuellement le ramasse-miettes, si c'est approprié (et il y a peu de cas où ça l'est),

attendez que tous les objets qui sont dans la file d'attente d'accessibilité aient appelé leurs finaliseurs, exécutez de nouveau le ramasse-miettes (pour balayer tous les objets qui viennent juste d'être finalisés). Ceci doit récupérer autant d'espace que possible avant de continuer l'exécution et peut être réalisé comme ceci :

```
System.GC.Collect;
System.GC.WaitForPendingFinalizers;
System.GC.Collect;
```

L'appel de la méthode *Collect* du ramasse-miettes sans aucun paramètre le force à parcourir chacune des trois générations et à récupérer l'espace de tous les objets inaccessibles du tas managé. Il existe une version surchargée de *Collect* qui prend en paramètre un nombre entier qui limite les générations à parcourir (de la génération zéro jusqu'à la génération spécifiée).

Vous pouvez trouver des schémas et d'autres informations sur la façon dont le ramasse-miettes travaille dans l'article en deux parties de Jeffrey Richter du magazine de MSDN :

Garbage Collection-Part 1: Automatic Memory Management in the Microsoft .NET Framework, Jeffrey Richter, MSDN Magazine, novembre 2000.
Garbage Collection-Part 2: Automatic Memory Management in the Microsoft .NET Framework, Jeffrey Richter, MSDN Magazine, décembre 2000.

3. Le pattern Dispose : Finalisation déterministe

Il y a des scénarios qui peuvent exiger le choix de la finalisation déterministe d'un objet (forcer la finalisation par le programmeur à un moment déterminé) :

1. Scénario 1
   Vous avez écrit une classe qui utilise des objets managés qui eux font usage de ressources non managées (par exemple des objets de la FCL tels que des connexions de base de donnée ou des fichiers), et il est normal qu'un client de ce type d'objet puisse libérer ces ressources contrôlées et encapsulées à un moment déterminé. En d'autres termes vous souhaitez exposer d'une façon ou d'une autre les finaliseurs des objets que vous employez au client de ces objets. Ceci pourrait être décrit comme une classe utilisant des ressources non managées encapsulées dans un objet.
2. Scénario 2
   Vous avez écrit une classe qui utilise des ressources non managées, et il normal qu'un client de ce type d'objet puisse libérer les ressources non managées à un moment déterminé. En d'autres termes vous souhaitez exposer d'une façon ou d'une autre votre finaliseur au client de l'objet. Ceci pourrait être décrit comme une classe utilisant des ressources non managées.

Le premier scénario est beaucoup plus fréquent que le second, bien que parfois une classe donnée s'adapte aux deux scénarios.

Dans les deux cas il est tout à fait envisageable que le programmeur imagine un certain système afin de permettre la finalisation déterministe d'un objet comme nous l'avons vu plus tôt. Toutefois à la place vous seriez avisé de suivre à cette fin le mécanisme proposé ici, qui est de mettre en application le *pattern Dispose*. Ce pattern définit formellement comment offrir la finalisation déterministe à un client d'objet, donnant une stabilité aux développeurs qui utiliseront vos objets.

Pour implémenter le *pattern Dispose*, votre objet doit implémenter l'interface ***System.IDisposable***. C'est une interface simple possédant un seul membre, une méthode sans paramètre appelée **Dispose**, il s'agit plus précisément d'une procédure. Quand une classe implémente **IDisposable**, elle propose la méthode publique **Dispose** comme moyen de libérer les

MCours.com

ressources non managées des objets qui sont directement ou indirectement utilisées par l'objet (cependant la mémoire de l'objet sera récupéré plus tard par le ramasse-miettes).

Quand les classes examinées implémentent le *pattern Dispose*, vous trouverez intéressant qu'elles offrent une méthode alternative, appelé **Close**, pour faire le même travail. C'est simplement une facilité offerte au programmeur qui utilisent ces objets, car il semble souvent plus approprié de fermer (close) certain type de ressource (tels que des fichiers ou des connexions de base de données) au lieu de les libérer. Notez que la méthode **Close** ne fait pas partie du *pattern Dispose*, c'est simplement un point d'entrée alternatif optionnel permettant d'appeler la méthode**Dispose**. Typiquement **Close** appellera simplement **Dispose**, donnant exactement le même résultat.

3-1. Implémenter IDisposable.Dispose

En implémentant le *pattern Dispose*, la méthode **Close** (si elle est présente) devrait être publique et non-virtuelle et devrait simplement appeler la méthode **Dispose**.

La méthode **Dispose** de l'interface ***IDisposable*** devrait libérer les ressources non managées possédées par l'objet (soit directement soit indirectement par d'autres objets) et devrait être implémentée de telle manière qu'elle puisse être appelée de nombreuses fois sans provoquer d'exception. De plus, **Dispose** devrait être publique et également scellée (**sealed**, en C#) ou**final** (en CIL [**C**ommon **I**ntermediate **L**anguage] ou en Delphi .NET) l'empêchant ainsi d'être surchargée dans des classes dérivées.

Le comportement des compilateurs du C# et de Delphi quand vous implémentez une méthode d'interface, est de s'assurer qu'elle est virtuelle (c'est une condition du CLR). Si la méthode est déclarée avec le modificateur **virtual**, alors les choses restent en l'état. Cependant une méthode qui ne serait pas déclarée

virtuelle sera compilée comme si elle avait été définie soit avec le modificateur **virtual** ou **sealed** (C#) ou encore **final** (Delphi). Ceci est fait automatiquement pour éviter des problèmes avec le polymorphisme dans les descendants, puisque l'ancêtre posséde une méthode virtuelle qui n'est pas censée l'être selon le code source.

En bref, les deux points suivants décrivent ce que vous devez faire pour implémenter le *pattern Dispose* pour les deux types différents de classes que vous pourriez avoir besoin d'écrire. Les sections suivantes donneront les détails et des exemples de code afin de clarifier ce point :

- Une classe correspondant au scénario 1, implémente **Dispose** et de là appelle les méthodes **Dispose/Close** des objets de la classe utilisant des ressources non managées.
- Une classe correspondant au scénario 2, implémente une méthode **Dispose** interne qui fait réellement le nettoyage. Elle implémente également ***IDisposable.Dispose*** et un finaliseur, ces deux méthodes appellent la routine interne **Dispose**, mais ***IDisposable.Dispose*** indique également au ramasse-miettes de ne pas appeler le finaliseur.

3-2. Le pattern Dispose et les wrappers de ressources non managées

Si vous écrivez une classe qui fait usage d'objets encapsulant des ressources non managées (tel que dans le scénario 1 ci-dessus), il est alors aisé d'implémenter le *pattern Dispose*. Vous implémentez **Dispose** pour appeler les méthodes **Close** ou **Dispose** de tous vos objets encapsulant des ressources non managées.

Employons une instance d'une classe simple qui utilise un

objet **FileStream** pour accéder à un fichier. **FileStream** est une classe qui utilise une ressource non managée (un handle de fichier) et implémente le *pattern Dispose*. Cependant, à la différence de la plupart des classes elle propose la méthode **Dispose** en tant que méthode protégée et vous propose seulement la méthode**Close** publique pour libérer la ressource.

Voici à quoi ressemble le *pattern Dispose* sous Delphi pour .NET :

```
//Répertoire : Destructeurs-
finaliseurs\Managed Code\Dispose
pattern\Normal
uses
  System.IO;

type
  TBaseResource = class(System.Object,
IDisposable)
  private
    // Trace si Dispose a déjà été appelé.
    disposed: Boolean;
    // Objet encapsulant une ressource non
managée
    fileStream: FileStream;
  public
    constructor Create(const NomDeFichier:
String);
    // Implémente IDisposable et s'assure que
IDisposable.Dispose ne peut être surchargée.
    procedure Dispose;
    procedure Close;
    procedure FaitqqChose;
  end;

constructor TBaseResource.Create(const
NomDeFichier: String);
begin
  inherited Create;
  WriteLn('Le constructeur TBaseResource
alloue une ressource non managée
encapsulée.');
  fileStream := System.IO.FileStream.Create(
```

```
NomDeFichier.Trim,System.IO.FileMode.Create,Sy
stem.IO.FileAccess.ReadWrite);

end;

procedure TBaseResource.Dispose;
begin
  // Vérifie si Dispose a déjà été appelé.
  if not disposed then
  begin
    WriteLn('Le pattern Dispose libère la
ressource encapsulant des ressources non
managées.');
    fileStream.Close;
    disposed := True;
  end
end;

procedure TBaseResource.Close;
begin
  Dispose;
end;

procedure TBaseResource.FaitqqChose;
begin
  WriteLn('Dans TBaseResource.FaitqqChose');
End;

var
  BR: TBaseResource;
begin
  BR :=
TBaseResource.Create('C:\Temp\TestFile.txt');
  try
    BR.FaitqqChose;
  finally
    if Assigned(BR) then
      BR.Dispose; // soit cet appel
      //BR.Close; // soit celui-ci
  end;
end.
```

Avec ce code qui crée l'objet, l'utilise puis appelle ses méthodes **Dispose** ou **Close**, on obtient ce résultat :

```
C:\WINDOWS\System32\cmd.exe

C:\temp>ResourceObjectEg.exe
Le constructeur TBaseResource alloue une ressource non-managée encapsulée.
Dans TBaseResource.FaitqqChose.
Le pattern Dispose libère la ressource encapsulant des ressources non-managées.

C:\temp>_
```

Maintenant que nous avons implémenté le *pattern Dispose*, le programmeur qui emploie cette classe à la possibilité de laisser la finalisation non déterministe habituelle avec la fermeture du fichier, puisque la classe **FileStream** fermera le fichier dans son finaliseur (C# destructeur). Cependant il peut également explicitement fermer le fichier en appelant la méthode **Dispose** ou**Close** si nécessaire. Notez que nous étudierons spécifiquement la façon d'employer des objets *disposable* dans une prochaine section.

Dans ces deux classes la méthode **Dispose** emploie un champ privé, ***disposed***, pour déterminer si elle a déjà été appelée. Ainsi, appeler **Dispose** ou **Close** plusieurs fois ne pose pas de problème. Cependant, cette implémentation du *pattern Dispose* n'est pas 'thread-safe'. Un autre thread pourrait commencer à libérer l'objet après que les ressources non managées encapsulées soient libérées, mais avant que le champ interne ***disposed*** soit assigné à **True**. Si vous écriviez une classe pour un usage au sein d'une application multi-thread et que vous vouliez vous assurer qu'elle sera 'thread-safe' voici comment vous le feriez pour la méthode **Dispose**. La modification suivante de **Dispose** remédie à cela en verrouillant l'objet pour la durée d'exécution de la méthode afin d'empêcher tout autre thread d'appeler **Dispose** au même moment.

Notez que Delphi à la différence du C# ne dispose pas du mot-clé *lock*, utilisez à la place ***System.Threading.Monitor*** dans un bloc **try/finally**.

```
//Répertoire : Destructeurs-
finaliseurs\Managed Code\Dispose
pattern\Thread-safe
```

```
uses
  System.IO,
  System.Threading;
...
procedure TBaseResource.Dispose;
begin
  // Sécurise ce traitement ('thread-safe').
  Monitor.Enter(Self);
  try
    // Vérifie si Dispose a déjà été appelée.
    if not disposed then
    begin
      WriteLn('Le pattern Dispose libère la
ressource encapsulant des ressources non
managées.');
      fileStream.Close;
      disposed := True;
    end
  finally
    Monitor.Exit(Self);
  end
end;
```

3-3. Destructeurs et IDisposable sous Delphi pour .NET

Si vous utilisez Delphi pour .NET, vous pouvez clairement implémenter ***IDisposable*** comme nous l'avons vu. Cependant, comme la plupart des développeurs employant Delphi pour .NET sont susceptibles d'avoir auparavant utilisé Delphi Win32, les ingénieurs de Borland R&D ont décidé de vous faciliter les choses dans le cas où vous utiliseriez des classes d'objet encapsulant des ressources. Le compilateur implémentera automatiquement ***IDisposable*** à votre place s'il voit que vous avez défini un destructeur correspondant exactement à la signature suivante (signature typique d'un destructeur Delphi) :

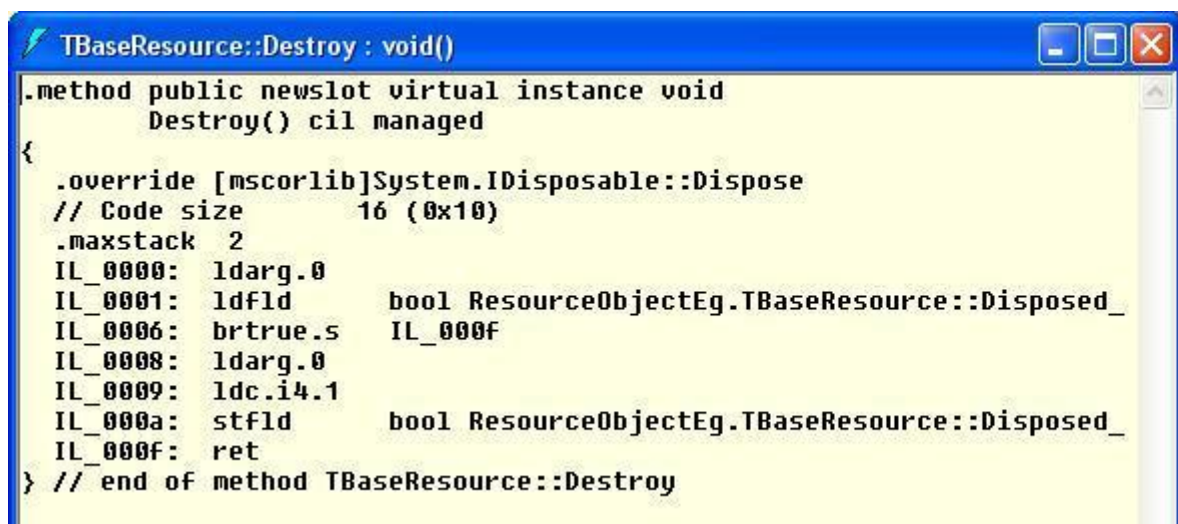
```
destructor Destroy; override;
```

Si vous avez un tel destructeur, la classe sera en interne marquée comme implémentant l'interface ***IDisposable*** (c'est une erreur

d'essayer de déclarer explicitement que ***IDisposable*** est implémentée dans la classe). Dans ce cas l'utilisation du désassembleur .NET, IL DASM, nous le confirme :

Analyse sous ILDASM du code généré

En outre, un destructeur avec cette signature sera marqué comme une implémentation de **IDisposable.Dispose** vous permettant de faire votre nettoyage dans un endroit familier. Ici IL DASM montre le code derrière un destructeur Delphi 'vide' et vous pouvez voir que tandis que la méthode s'appelle **Destroy**, c'est une surcharge de **IDisposable.Dispose** :



Méthode Destroy 'vide'

Le code **IL** précédent correspond au code Delphi suivant :

```
destructor TBaseResource.Destroy;
begin
 If Disposed_=True
  then Exit;
 ...
 Disposed_:=True ;
end;
```

On découvre donc la création, dans la classe **TBaseRessource**, d'une variable interne nommée ***Disposed_*** :

Attention ce champ est ajouté à partir du moment où vous déclarez une méthode Destroy et seulement dans ce cas.

Ce raccourci, l'implémentation automatique de **IDisposable.Dispose**, est tout à fait pratique pour les objets qui utilisent des objets encapsulant des ressources non managées. Votre destructeur appelle les méthodes **Dispose** des objets wrapper et le client de votre objet appelle le destructeur (provoquant la finalisation déterministe) ou pas (permettant la finalisation non déterministe). De plus, la présence du destructeur sera utile lors du portage du code sous .NET, mais aussi lorsque vous essaierez d'écrire du code multi-plateforme afin qu'il fonctionne sous Win32 (compilé avec Delphi), Linux (compilé avec Kylix) et NET (compilé avec Delphi pour .NET).

Pour aller plus loin dans la compréhension de l'implémentation de la méthode **Free**, elle vérifie si l'interface ***IDisposable*** est implémenté, et si oui, elle appelle pour vous la méthode **Dispose**de l'interface . Si vous suivez la signature du destructeur ci-dessus alors **Free** appellera le destructeur, mais si vous implémentez **Dispose** comme nous l'avons fait dans la section précédente dans ce cas votre méthode **Dispose** sera appelée.

Ce que tout ceci signifie c'est que vous pouvez implémenter la classe en suivant les lignes de code suivantes. Notez que la classe ne requiert pas l'implémentation de ***IDisposable***; cela se fait dans les coulisses et produirait une erreur du compilateur si vous le faisiez.

```
// Destructeur-finaliseurs\Managed Code\Dispose pattern\Destructor

{$APPTYPE CONSOLE}

uses
  System.Threading,
  System.IO;

type
  TBaseResource = class
  private
    // la déclaration du code suivant n'est plus nécessaire sous Delphi 2005
    // le compilateur ajoute et gére la variable Disposed_ dans l'implémentation
de Destroy
    // disposed: Boolean;

    // Objet encapsulant une ressource non managée
    fileStream: FileStream;
  public
    constructor Create(const NomDeFichier: String);
    destructor Destroy; override;
    procedure Close;
    procedure FaitqqChose;
  end;

constructor TBaseResource.Create(const NomDeFichier: String);
begin
  inherited Create;
```

```
  WriteLn('Le constructeur TBaseResource alloue une ressource non managée
encapsulée.');
  fileStream :=
System.IO.FileStream.Create(NomDeFichier.Trim,System.IO.FileMode.Create,System.I
O.FileAccess.ReadWrite);
end;

destructor TBaseResource.Destroy;
begin
  // Sécurise ce traitement ('thread-safe').
  Monitor.Enter(Self);
  try
   begin
    WriteLn('Le destructeur Destroy (pattern Dispose) libère la ressource
encapsulant des ressources non managées.');
    fileStream.Close;
   end;
  finally
    Monitor.Exit(Self);
  end;
end;


procedure TBaseResource.Close;
begin
  Free;
end;

procedure TBaseResource.FaitqqChose;
begin
  WriteLn('Dans TBaseResource.FaitqqChose');
end;

var
  BR: TBaseResource;
begin
  BR := TBaseResource.Create('C:\Temp\TestFile.txt');
  try
    BR.FaitqqChose;
  finally
    if Assigned(BR) then
      BR.Free;
  end;
end.
```

Le code précédent (Delphi 2005) diffère de l'article original (Delphi 8), la méthode **Destroy** est exécuté si la variable interne ***Disposed_*** est à **False**.

```
C:\WINDOWS\System32\cmd.exe

C:\temp>ResourceObjectEg.exe
Le constructeur TBaseResource alloue une ressource non-managée encapsulée.
Dans TBaseResource.FaitqqChose
Le destructeur Destroy (pattern Dispose) libére la ressource encapsulant des ressources non-managées.

C:\temp>
```

Résultat

Ce code emploie la classe **Monitor** pour s'assurer du fonctionnement dans un environnement multi-thread, toutefois ce ne sera pas nécessaire dans la version commerciale de Delphi pour .NET. Le compilateur générera automatiquement ce code aussi bien que la déclaration et l'utilisation du flag déjà mentionné (***disposed*** dans le code d'exemple ci-dessus).

La prise en charge par le compilateur du code 'thread-safe' ne semble pas implémentée sous Delphi 2005.

```
//Répertoire : Destructeurs-finaliseurs\Managed Code\Dispose pattern\Destructor,
well written
```

```
uses
  System.IO;

type
  TBaseResource = class
  private
    // Objet encapsulant une ressource non managée
    fileStream: FileStream;
  public
    constructor Create(const NomDeFichier: String);
    destructor Destroy; override;
    procedure Close;
    procedure FaitqqChose;
  end;

constructor TBaseResource.Create(const NomDeFichier: String);
begin
  inherited Create;
  WriteLn('Le constructeur TBaseResource alloue une ressource non managée
encapsulée');
  fileStream := System.IO.FileStream.Create(
    NomDeFichier, FileMode.Open, FileAccess.Read, FileShare.Read);
end;

destructor TBaseResource.Destroy;

 // Code pour Delphi 8 ?
 // Pour Delphi 2005 l'exemple de code précédent n'a pas besoin d'être modifié.

begin
  WriteLn('Le destructeur Destroy (pattern Dispose) libère la ressource
encapsulant des ressources non managées.');
  fileStream.Close;
end;

procedure TBaseResource.Close;
begin
  Free;
end;

procedure TBaseResource.FaitqqChose;
begin
  WriteLn('Dans TBaseResource.FaitqqChose');
end;
```

Nous examinerons sous peu la syntaxe d'utilisation d'un objet avec un destructeur en Delphi .NET, mais vous devriez noter la différence entre un destructeur C# et un destructeur Delphi, qui est d'une importance particulière si vous êtes amené à écrire dans ces deux langages. Un destructeur C# implique la génération automatique d'un finaliseur par le compilateur, tandis qu'un destructeur Delphi (avec la bonne signature) provoque l'implémentation automatique de l'interface ***IDisposable*** par le compilateur.

"L'équipe CLR" de Microsoft a conseillé aux ingénieurs de Borland R&D de ne pas faire faire à leur compilateur des générations automatiques trop nombreuses de finaliseurs car, s'ils sont corrects sur une petite échelle, c'est-à-dire codés manuellement, sur une grande échelle (comme quand ils sont générés automatiquement par le compilateur) les finaliseurs peuvent faire chuter le système CLR en entier. Relisez le point suivant : *Finaliseurs sous Delphi .NET*, si vous souhaitez réaliser quelque chose sans employer de finaliseur.

3-4. Le pattern Dispose et les ressources non managées

La dernière section (*Le pattern Dispose et les wrappers de ressources non managées*) a exploré comment employer le *pattern Dispose* quand votre classe a affaire avec des objets qui encapsulent les complexités de traitement des objets non managés, qui est de loin le scénario le plus susceptible de se produire en écrivant du code pour .NET. Cette section examine le cas où votre classe utilise directement une ressource non managée, exigeant de ce fait un finaliseur, et voit quel impact ceci a sur l'exécution du *pattern* *Dispose*.

Ce scénario est un peu plus compliqué. Si la méthode **Dispose** est appelée, elle doit libérer les ressources non managées qui sont normalement libérées par le finaliseur (rendant le finaliseur effectivement superflu). Une fois ceci fait, elle devrait également informer le ramasse-miettes ne pas appeler le finaliseur. Ceci est réalisé en passant l'objet comme paramètre à la méthode**System.GC.SuppressFinalize** et rend la libération de l'objet lors de l'opération de nettoyage bien plus efficace (l'objet n'est jamais placé dans la file d'attente d'accessibilité).

3-5. Surcharge de Dispose

Puisque les ressources de l'objet peuvent maintenant être libérées soit par la méthode **Dispose** si elle est appelée, soit par le finaliseur sinon, une implémentation ordinaire peut compter sur l'utilisation d'une autre version de la méthode **Dispose**, celle-ci étant inaccessible au client de l'objet et prenant un paramètre booléen pour indiquer d'où elle est appelée. Cette nouvelle méthode **Dispose** devrait être protégée et virtuelle ainsi les classes dérivées ne pourront étendre son comportement.

L'approche commune est que quand la méthode publique, et sans paramètre, **Dispose** appelle cette version protégée, elle lui passe *True* indiquant que la libération est provoquée par le code d'un utilisateur. Ceci signifie qu'il est sûr (safe) d'accéder aux objets finalisables référencés par des champs puisque leurs finaliseurs n'auront pas encore été appelés, comme pour des ressources non managées. Quand les finaliseurs appellent **Dispose** ils lui passent *False*, pour indiquer qu'elle est appelée par le thread finaliseur du CLR, et ainsi seulement des ressources non managées possédées par l'objet peuvent être libérées.

Voici un exemple d'une implémentation typique dans une classe simple :

```
// Destructeurs-finaliseurs\Managed Code\Dispose pattern and finalizer\Normal

{$APPTYPE CONSOLE}

{$DEFINE LiberationDispose}
{DEFINE LiberationFree}
{DEFINE PasDeLiberation}

{$WARN UNIT_PLATFORM OFF}
uses Borland.Vcl.Windows; // Provoque un avertissement concernant la plate-
forme.

type
  TBaseResource = class(System.Object, IDisposable)
  private
     // Trace si Dispose a été appelé.
     // Pas de déclaration de Destroy, donc c'est au programmeur de le gérer.
    disposed: Boolean;
    // La ressource est un handle
    handle: IntPtr;
  strict protected
    procedure Finalize; override;
    procedure Dispose(disposing: Boolean); overload; virtual;
  public
    constructor Create;
    // Implémente IDisposable et s'assure que IDisposable.Dispose ne peut être
surchargée.
    procedure Dispose; overload;
    procedure Close;
    procedure FaitqqChose;
  end;

constructor TBaseResource.Create;
begin
  inherited Create;
  handle := IntPtr.Zero;
  WriteLn('Le constructeur de TBaseResource devrait avoir le code nécessaire
pour allouer la ressource.');
end;
```

```
procedure TBaseResource.Finalize;
begin
  try
    WriteLn('Le finaliseur de TBaseResource libère la ressource.');
    Dispose(false);
  finally
    inherited;
  end;
end;

procedure TBaseResource.Dispose;
begin
  WriteLn('Le pattern Dispose est utilisé pour libérer la ressource.');
  Dispose(true);
  // Dispose est appelé par le programmeur dans ce cas
  // on empêche l'appel du finaliseur par le ramasse-miettes (GC)
  GC.SuppressFinalize(Self);
end;

procedure TBaseResource.Close;
begin
  Dispose;
end;

procedure TBaseResource.Dispose(disposing: Boolean);
begin
   // Vérifie si Dispose a déjà été appelée.
  if not disposed then
  begin
    if disposing then
    begin
     WriteLn(#9+'Libére les ressources managées.');
    end;
    WriteLn(#9+'Libére les ressources non managées.');
    if handle <> IntPtr.Zero then
    begin
      CloseHandle(handle.ToInt32);
      WriteLn('Réassigne le handle avec une valeur sûre.');
      handle := IntPtr.Zero;
    end;
    disposed := True;
  end;
end;

procedure TBaseResource.FaitqqChose;
begin
  WriteLn('Dans TBaseResource.FaitqqChose.')
end;

var
  BR: TBaseResource;
begin
 {$ifdef PasDeLiberation}
  BR := TBaseResource.Create;
  BR.FaitqqChose;
{$endif}

{$ifdef LiberationDispose}
  BR := TBaseResource.Create;
  try
    BR.FaitqqChose;
  finally
    BR.Dispose;
  end;
{$endif}

{$ifdef LiberationFree}
  BR := TBaseResource.Create;
  try
    BR.FaitqqChose;
  finally
```

```
    BR.Free;
  end;
{$endif}
end.
```

```
Invite de commandes

c:\temp>ResourceObjectEg.exe
Le constructeur de TBaseResource devrait avoir le code nécessaire pour allouer la ressource.

Dans TBaseResource.FaitqqChose.
Le pattern Dispose est utilisé pour libérer la ressource.
        Libére les ressources managées.
        Libére les ressources non-managées.

c:\temp>
```

Résultat

Comme mentionné ci-dessus, quand vous implémentez une méthode d'une interface qui n'est pas déclarée avec la directive **virtual**, le compilateur Delphi pour .NET la traitera comme *virtual* et*final* ainsi vous n'avez pas besoin d'employer la directive *final* pour cette méthode. IL DASM le confirme :

```
TBaseResource::Dispose : void()

.method public hidebysig newslot virtual final
        instance void  Dispose() cil managed
{
  .override [mscorlib]System.IDisposable::Dispose
  // Code size       40 (0x28)
  .maxstack  3
  IL_0000:  ldsfld     class Borland.Delphi.Text Borland.Delphi.TextOutput::Output
  IL_0005:  ldstr      bytearray (4C 00 65 00 20 00 70 00 61 00 74 00 74 00 65 00
                                  72 00 6E 00 20 00 44 00 69 00 73 00 70 00 6F 00
                                  73 00 65 00 20 00 65 00 73 00 74 00 20 00 75 00
                                  74 00 69 00 6C 00 69 00 73 00 E9 00 20 00 70 00
                                  6F 00 75 00 72 00 20 00 6C 00 69 00 62 00 E9 00
                                  72 00 65 00 72 00 20 00 6C 00 61 00 20 00 72 00
                                  65 00 73 00 73 00 6F 00 75 00 72 00 63 00 65 00
                                  2E 00 )
  IL_000a:  ldc.i4.0
  IL_000b:  call       instance class Borland.Delphi.Text Borland.Delphi.Text::Write

  IL_0010:  call       instance void Borland.Delphi.Text::WriteLn()
  IL_0015:  call       void Borland.Delphi.Units.System::@_IOTest()
  IL_001a:  ldarg.0
  IL_001b:  ldc.i4.1
  IL_001c:  callvirt   instance void ResourceObjectEg.TBaseResource::Dispose(bool)
  IL_0021:  ldarg.0
  IL_0022:  call       void [mscorlib]System.GC::SuppressFinalize(object)
  IL_0027:  ret
} // end of method TBaseResource::Dispose
```

IL DASM virtual-final

Cette implémentation est typique de celles que vous trouvez dans des tutoriaux sur la programmation .NET, mais elle n'est pas 'thread-safe'. Un autre thread pourrait commencer à libérer l'objet après que les ressources contrôlées soient libérées, mais avant que le champ interne ***disposed*** soit assigné à *True*. La modification suivante de la méthode protégée **Dispose** remédie à ceci en verrouillant l'objet pendant l'exécution de la méthode **Dispose** afin d 'empêcher tout autre thread d'appeler **Dispose**.

```
//Répertoire : Destructeurs-finaliseurs\Managed Code\Dispose pattern and
finalizer\Thread-safe
uses
  System.Threading,
  Borland.Win32.Windows;
...
```

```
procedure TBaseResource.Dispose(disposing: Boolean);
begin
   // Sécurise ce traitement ('thread-safe').
  Monitor.Enter(Self);
  try
     // Vérifie si Dispose a déjà été appelée.
    if not disposed then
    begin
      if disposing then
      begin
       WriteLn(#9+'Libére les ressources managées.');
      end;
      WriteLn(#9+'Libére les ressources non managées.');
      if handle <> IntPtr.Zero then
      begin
        CloseHandle(handle.ToInt32);
        WriteLn('Réassigne le handle avec une valeur sûre.');
        handle := IntPtr.Zero;
      end;
      disposed := True;
    end;

  finally
    Monitor.Exit(Self);
  end;
end;
```

Nous avons vu plus tôt le pattern destructeur spécial de Delphi, qui est traduit par le compilateur par une implémentation implicite de ***IDisposable***. Il devrait être clair ici que ce pattern n'est pas applicable quand vous avez un finaliseur à implémenter. Le style de codage recommandé par Borland est de ne pas mélanger les destructeurs traditionnels aux finaliseurs du CLR. Si vous avez besoin d'un finaliseur dans votre classe, vous devriez implémenter complétement ***IDisposable***, comme nous l'avons fait ici. Ne mélangez pas les **finaliseurs** au destructeur spécial**Destroy**, car il n'est pas garanti qu'il fonctionne à l'avenir.

Il est actuellement possible d'outre passer cette indication, mais vous ne gagnez rien en faisant ainsi. À l'avenir le compilateur peut bien interdire l'implémentation d'un finaliseur en association avec le pattern destructeur spécial.

3-6. Considérations Post-Dispose

Nous avons vu de nombreuses informations au sujet du *pattern Dispose*, mais nous n'en avons pas encore tout à fait terminé avec lui. Tandis que **Dispose** doit pouvoir être appelé de multiples fois, sans déclencher d'exception, une fois que l'objet a été libéré (soit par le finaliseur soit par un appel à **Dispose**) il devrait être considéré comme inutilisable puisque ses ressources principales ont été libérées. Pour imposer ce caractère inutilisable après un appel à **Dispose**, il est vivement recommandé que dans ce cas les méthodes normales déclenchent une exception**System.ObjectDisposedException** .

La méthode ***FaitqqChose*** insignifiante dans notre classe devrait ressembler à ceci

```
//Répertoire : Destructeurs-finaliseurs\Managed Code\Dispose pattern and
finalizer\Thread-safe, well written
...
procedure TBaseResource.FaitqqChose;
begin
  if disposed then
    raise ObjectDisposedException.Create(ToString);
  WriteLn('Dans TBaseResource.FaitqqChose')
end;
```

3-7. Employer un objet disposable

Vous pourriez utiliser un objet qui implémente le *pattern Dispose* de la manière suivante :

```
var
  BR: TBaseResource;
begin
  BR := TBaseResource.Create;
  try
    BR.FaitqqChose;
  finally
    if Assigned(BR) then
      BR.Dispose;
  end;
end.
```

Ceci fonctionne bien généralement, toutefois quelques objets ne rendent pas leurs méthodes **Dispose** disponibles pour les clients des objets (tels que la classe **FileStream**). Dans ces cas-là vous pouvez appeler la méthode **Close** si vous le souhaitez, mais si vous voulez une certaine uniformité d'utilisation de vos objets qui implémentent le *pattern Dispose* vous pourriez y accéder par l'intermédiaire de l'interface ***IDisposable***.

```
var
  BR: TBaseResource;
begin
  BR := TBaseResource.Create;
  try
    BR.FaitqqChose;
  finally
    if Assigned(BR) then
      (BR as IDisposable).Dispose;
  end;
end.
```

Toutefois grâce à la manière dont a été implémenté **TObject.Free**, vous pouvez également vous servir d'un objet disposable comme ceci :

```
var
  BR: TBaseResource;
begin
  BR := TBaseResource.Create;
  try
    BR.FaitqqChose;
  finally
    BR.Free;
  end;
end.
```

Ce comportement a été conçu pour aider les développeurs Delphi à porter leur code sur la plate-forme .NET sans devoir réécrire d'interminables appels à **Free**. Notez que si vous portez du code sous .NET et constatez que vous n'avez aucun besoin de faire de finalisation (tous les destructeurs sont des appels à **Free**, et ceux-ci ne contrôlent pas de ressources non managées), et qu'alors vous pouvez retirer le destructeur (ou employer la compilation conditionnelle pour l'empêcher d'être compilé), vous pouvez employer **Free** sans problème. Ceci peut entraîner une légère surcharge mais elle devrait être négligeable.

3-8. En résumé

Cet article vous a permis d'approfondir les problématiques relatives aux destructeurs et aux finaliseurs en rapport avec le **garbage collector**, qui prend désormais la responsabilité de libérer la mémoire occupée par les objets qui ne sont plus en service. De nombreux détails sont à prendre en compte, mais après avoir lu cet article vous devriez garder à l'esprit les points suivants :

- N'implémentez un finaliseur (ou un destructeur en C#) que si vous avez des ressources non managées à libérer. Rappelez-vous qu'aucun objet référencé par votre objet n'a besoin d'être libéré ; le ramasse-miettes le fera.
- Dans la mesure du possible utilisez les classes .NET existantes pour l'accès aux ressources non managées (telles que des handles de fichiers, handles de socket, handles de fenêtre ou des connexions de base de données) au lieu d'implémenter un finaliseur dans une nouvelle classe.
- Les finaliseurs ne s'exécutent en aucune manière de façon rapide ni dans un ordre prévisible, ils ajoutent une surcharge de travail lors de la construction d'objets et lors de l'opération de nettoyage et font persister vos objets en mémoire beaucoup plus longtemps que vous pouviez le prévoir (ce qui fait que tous les objets référencés par l'objet finalisable existent beaucoup plus longtemps que vous l'aviez prévu).
- Implémentez le *pattern Dispose* pour permettre aux clients des objets de libérer les ressources non managées que vous utilisez directement ou indirectement. Si vous avez un finaliseur, il effectuera plus efficacement la récupération de votre objet et de ses ressources.
- Si vous implémentez le *pattern Dispose*, assurez-vous que vos méthodes déclenchent l'exception **System.ObjectDisposedException** si l'objet a déjà été nettoyé.
- Si vous implémentez le *pattern Dispose* mais aussi un finaliseur, assurez-vous que la méthode **Dispose** appelle la méthode **GC.SuppressFinalize**.
- Suivez un des patterns d'implémentation du *pattern Dispose* comme développé précédemment, selon le type de classe que vous implémentez (cf. Le *pattern Dispose : Finalisation déterministe*).
- Si vous projetez d'employer implicitement le *pattern Dispose* pour des objets sous Delphi pour .NET, soyez sûr que vous n'implémentez pas un finaliseur en tant que tel (si vous avez besoin d'un finaliseur, implémentez vous-même ***IDisposable***).

Table des matières

1. Public concerné

  1-1. Les sources

2. Destructeurs d'objet et Finaliseurs

  2-1. Introduction

  2-2. La destruction déterministe

  2-3. Destructeur sous Delphi Win32

  2-4. Commentaires sur les destructeurs

  2-5. Finalisation non déterministe

  2-6. Finaliseurs

  2-7. Quelques questions au sujet d'un finaliseur

  2-8. Finaliseurs sous C#

  2-9. Finaliseurs sous Delphi .NET

  2-10. Détails sur le Garbage Collector (ramasse-miettes)

    2-10-1. Algorithme de génération

    2-10-2. D'autres questions sur le finaliseur

3. Le pattern Dispose : Finalisation déterministe

  3-1. Implémenter IDisposable.Dispose

  3-2. Le pattern Dispose et les wrappers de ressources non managées

  3-3. Destructeurs et IDisposable sous Delphi pour .NET

  3-4. Le pattern Dispose et les ressources non managées

  3-5. Surcharge de Dispose

  3-6. Considérations Post-Dispose

  3-7. Employer un objet disposable

  3-8. En résumé

MCours.com