Les DLL

ActiveX

NB : ce fascicule fait partie d'un travail de diplôme sur le sujet « Technologie ActiveX & Visual Basic 6 ».
Les autres fascicules peuvent être demandés à fcomte@caramail.com.

Critiques, remarques, questions ? fcomte@caramail.com

1 - Caractéristiques

Une DLL ActiveX est une collection d'un ou de plusieurs modules de classe Visual Basic. Elle utilise la même structure de programmation que celle des modules de classes (rappelons qu'un module de classe est un modèle à partir duquel un objet est créé, et qui permet de définir des méthodes, des propriétés et des événements qui seront utilisées par d'autres parties de l'application). Les quelques différences mineures résident dans les propriétés associées au projet ActiveX. Une DLL ActiveX ne stocke pas seulement un module de classe, mais peut également accueillir des fonctions et des routines, des feuilles, des contrôles, des pages de propriétés ainsi que tout ce qu'il est possible d'inclure dans un programme Visual Basic. Une seule exception : les feuilles MDI (Multiple Document Interface).

On peut donc résumer ceci en disant qu'une DLL ActiveX n'est qu'un fourre-tout pour tout le code Visual Basic qu'on aimerait insérer dans une librairie standard. Néanmoins, il faut se souvenir de deux choses :

→ Une DLL ActiveX ne peut être exécutée directement.

→ Si nous n'avons pas besoin de partager votre code avec une autre application, pourquoi vouloir le placer dans une librairie ?

2 - Intérêt des DLL ActiveX

Très souvent, les programmeurs rassemblent un ensemble de routines qu'ils emploient à plusieurs reprises encore et encore lorsqu'ils écrivent leurs programmes. Certaines de ces routines peuvent être très complexes; d'autres peuvent implémenter des fonctions cachées de Windows. Nous pouvons employer certaines de ces routines souvent, d'autres rarement.

La plupart des programmeurs copient ces routines du dernier programme qui les a employés ou partagent un fichier source entre les applications. Ces approches ont plusieurs inconvénients.

Généralement, quand on copie le code d'un programme à un autre, on y opère invariablement des changements. Souvent, ces changements sont nécessaires pour que la routine puisse travailler dans notre nouvelle application, mais parfois ces changements optimisent simplement la routine. À moins qu'on ne propage nos améliorations de nouveau au programme original, elles peuvent être perdues.

Considérons également les problèmes que l'on rencontre si on trouve une erreur dans la routine. Nous devrons localiser chaque programme qui a contenu la routine et faire la même correction. Ceci peut être très long, particulièrement si nous avons changé la routine quand nous l'avons copiée de programme en programme.

Employer une librairie commune est également dangereux : les routines sont placées dans notre programme, mais à moins que nous ne prenions des mesures spéciales pour empêcher des changements à celles-ci, il est possible qu'un programmeur détruise le travail effectué par d'autres. On peut changer la routine dans la librairie commune pour régler un problème provenant d'un programme, et introduire par distraction un nouveau bug dans un programme différent.

Ainsi, une meilleure façon de partager du code est de prendre un peu de temps afin de créer sa propre DLL ActiveX. On peut alors déplacer les routines communes dans cette DLL et simplement partager cette dernière. Comme la DLL est codée « objet », elle peut facilement être distribuée et mise à jour de façon indépendante. Les mécanismes COM nous forcent à créer et à maintenir une interface bien définie : Visual Basic peut détecter quand cette interface a changé et ainsi nous avertir d'un problème à venir.

Les DLL ActiveX présentent néanmoins deux inconvénients. Le premier est une performance assez basse, ce qui n'affecte finalement pas trop la plupart des programmes. Le second est que nous sommes obligé de bien réfléchir quant à la façon dont nous allons développer nos fonctions, bien plus que si on les incluaient directement dans notre programme.

2.1 Avantages

- → Le code est facilement partagé entre les applications.
- → Les DLL offrent des performances excellentes, étant donné leur nature de composant « in process ».
- → La correction d'un bug dans une DLL ne requiert de la part du développeur que la distribution de la nouvelle DLL. Les applications utilisant les DLL sont automatiquement « corrigées ».

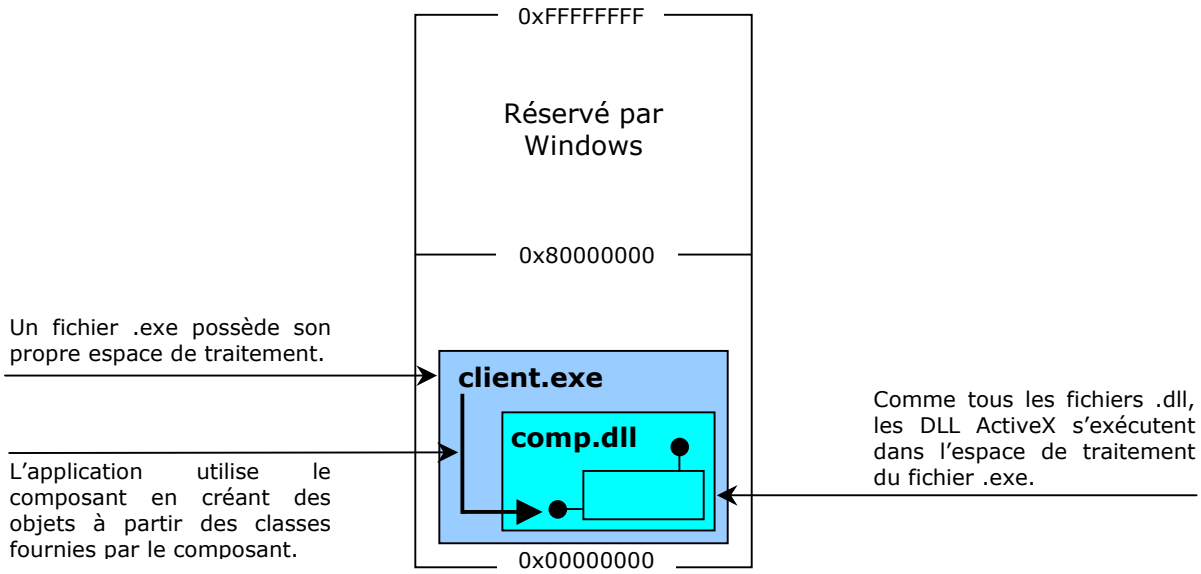
2.2 Inconvénients

- → Si une DLL d'une version donnée n'est pas compatible avec son prédécesseur, on doit revoir toutes les applications qui utilisent la DLL.
- → Une DLL ne peut créer de nouveaux threads d'exécution.
- → La complexité du déploiement de l'application est augmentée.
- → L'enregistrement, la vérification de version, ainsi que la vérification des composants sont requis pour une distribution sûre du produit.

En résumé, une DLL est idéale pour implémenter des objets standards que l'on va réutiliser ou partager avec d'autres applications. C'est également la solution pour définir des interfaces implémentées par d'autres objets, mais aussi le moyen de créer des objets de hautes performances ne possédant pas d'interface utilisateur.

3 - Une DLL ActiveX est un composant In-Process

Les serveurs in-process sont chargés dans le processus du client du fait qu'ils sont implémentés sous forme de DLL. L'avantage principal de ces serveurs est leur vitesse. Comme les objets sont chargés in-process, aucun changement de contexte n'est nécessaire pour accéder à leurs services, tout comme avec les DLL. Leur seul désavantage potentiel est que, comme le serveur in-process est en fait une DLL et non un exécutable potentiel (EXE), on ne peut s'en servir que dans le contexte d'un programme appelant ; il ne peut donc pas être exécuté en tant qu'application autonome.
Les DLL ActiveX - tout comme les contrôles ActiveX - sont implémentés sous forme de serveurs in-process.
La figure suivante montre un client appelant un composant in-process.



4 Création d'une DLL ActiveX

Microsoft propose dans la MSDN la création complète d'un exemple de DLL ActiveX (projet ThingDemo). C'est celui que nous utiliserons, dans la mesure où il passe complètement en revue la conception d'un serveur in-process, les références circulaires, la durée de vie des objets, les objets globaux, et le débogage. Nous créerons une classe qui permettra de visualiser dans la fenêtre d'exécution le mécanisme de création / destruction d'instances d'objet.

Contenu de l'exemple

Nous allons parcourir les sujets suivants :

4.1 Création du projet `ThingDemo`
Décrit comment définir des options de projet pour un composant in-process.

4.2 Création des propriétés et des méthodes pour la classe `Thing`
Ajoute deux propriétés et une méthode.

4.3 Ajout d'un code pour les événements `Initialize` et `Terminate`
Gestion du démarrage et de la terminaison du composant.

4.4 Création du projet test `TestThing`
Crée un groupe de projets composé de `ThingDemo` et d'un projet test pour que nous puissions déboguer le projet DLL in-process.

4.5 Création et test des objets `Thing`
Ajoute un code au projet `TestThing` pour créer des instances de l'objet `Thing` et pour appeler ses propriétés et ses méthodes.

4.6 Premier test : exécution de l'application test `TestThing`
Démontre le cycle de vie d'un objet fourni par un composant in-process.

4.7 Références circulaires et durée de vie des objets
Explore plus avant les cycles de vie des objets, en illustrant les effets des références circulaires sur l'arrêt des composants.

4.8 Ajout d'une feuille au projet `ThingDemo`
Ajoute du code pour afficher une feuille comme boîte de dialogue modale ou non modale, en utilisant la classe globale pour contrôler la feuille.

4.9 Utilisation de l'objet global dans `TestThing`
Teste l'objet global et les boîtes de dialogue, et donne plus de détails sur le débogage des DLL in-process.

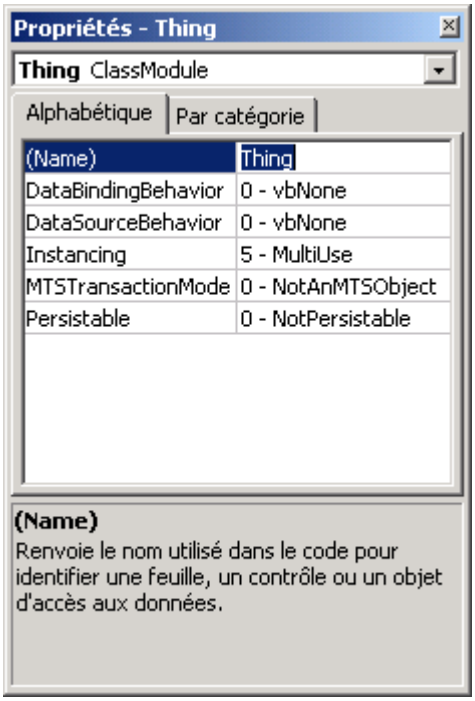
4.10 Compilation et test de la DLL `ThingDemo`
Compile le projet. Décrit comment tester le fichier .dll avec le projet test et comment l'inclure dans un autre projet.

4.11 Encore plus de références circulaires et arrêt des composants
Décrit la manière dont Visual Basic décharge un composant in-process après que le client ait libéré toutes les références à ses objets. Explique aussi comment les références circulaires peuvent empêcher ce déchargement.

4.1 Création du projet ThingDemo

Dans Visual Basic, la conception d'une DLL ActiveX débute par l'établissement des propriétés du projet.

Dans la fenêtre des propriétés de la classe nouvellement créée, modifions tout d'abord son nom en « `Thing` ». Détaillons la fenêtre des propriétés d'une classe DLL ActiveX.



« `Thing` » est donc le nom du module de classe à partir duquel nous pourrons créer des objets.

Les propriétés `DataBindingBehavior` et `DataSourceBehavior` sont relatives aux bases de données; `MTSTransactionMode` permet de diriger la manière avec laquelle les objets travailleront sous un serveur transactionnel COM+. Ne nous en soucions donc pas pour le moment.

Mais voyons plus en détail la propriété **`Instancing`** : elle détermine le degré de disponibilité d'un objet COM, c'est-à-dire si la classe est privée (et donc réservée au composant), ou si elle est mise à disposition de toutes les autres applications. Les valeurs sont les suivantes :

1 – `Private` : Même si l'objet possède des membres publics, aucun programme extérieur à la DLL ne peut y accéder, ni créer de nouvelles instances de la classe.

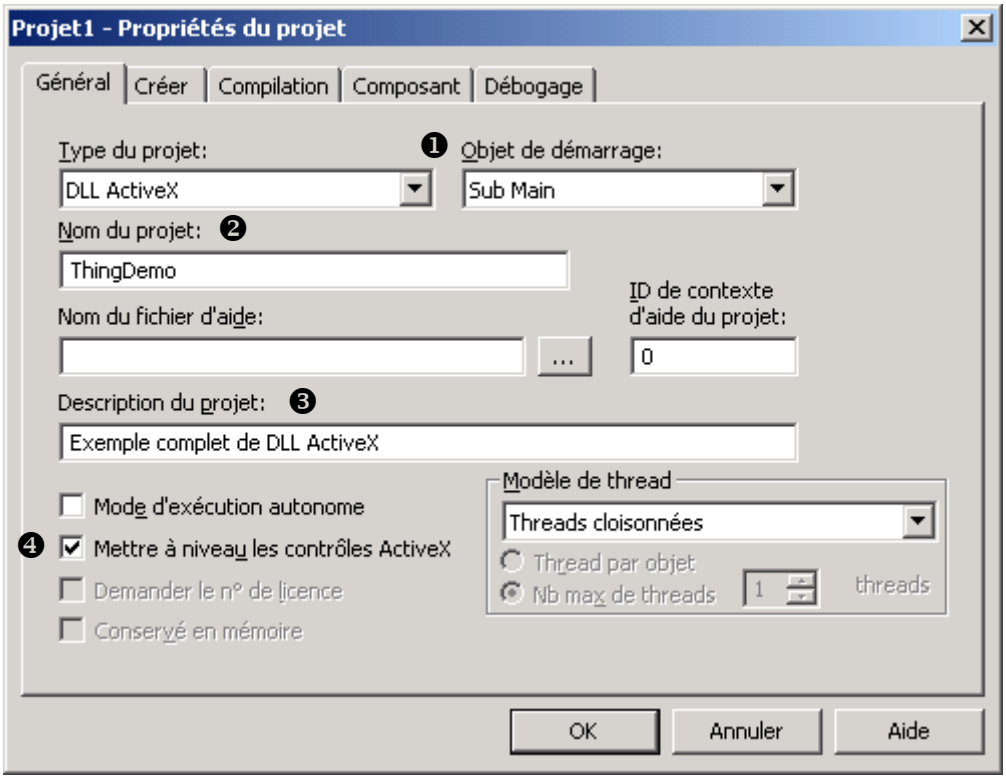
2 – `PublicNotCreatable` : Des programmes peuvent utiliser l'objet, mais pas le créer (`CreateObject` ou `New`). Utile si l'on veut retourner des objets contenant des données à un programme « conteneur », sans que ce dernier puisse créer des instances de l'objet.

5 – `MultiUse` : D'autres programmes peuvent créer des instances de la classe et les utiliser. C'est la valeur par défaut dans Visual Basic.

6 – `GlobalMultiUse` : Semblable à `MultiUse`, excepté que les propriétés et les méthodes de la classe peuvent être utilisées sans avoir à créer une instance de l'objet en premier. VB va créer une instance la première fois qu'elle sera utilisée. Cette méthode est déconseillée dans la mesure où on peut facilement être dépassé par le nombre d'instances créées sans que l'on ne sans aperçoive.

Et finalement, la propriété **`Persistable`** nous permet de décider si nous voulons que l'objet garde ses propriétés entre chaque nouvelle instance. Si cette propriété est validée (valeur 1), les événements `InitProperties`, `ReadProperties`, et `WriteProperties`, ainsi que la méthode `PropertyChanged` seront ajoutées à la classe. Ces outils travaillent en collaboration avec l'objet `PropertyBag`, afin de sauvegarder les informations entre chaque nouvelle instanciation de l'objet.

Voyons maintenant les propriétés du projet lui-même (Projet → Propriétés de Projet1…).



❶ Nous devons spécifier que l'objet de démarrage sera la procédure `Sub Main`, car on initialisera la DLL. Par défaut, cette zone indique « Aucun ».

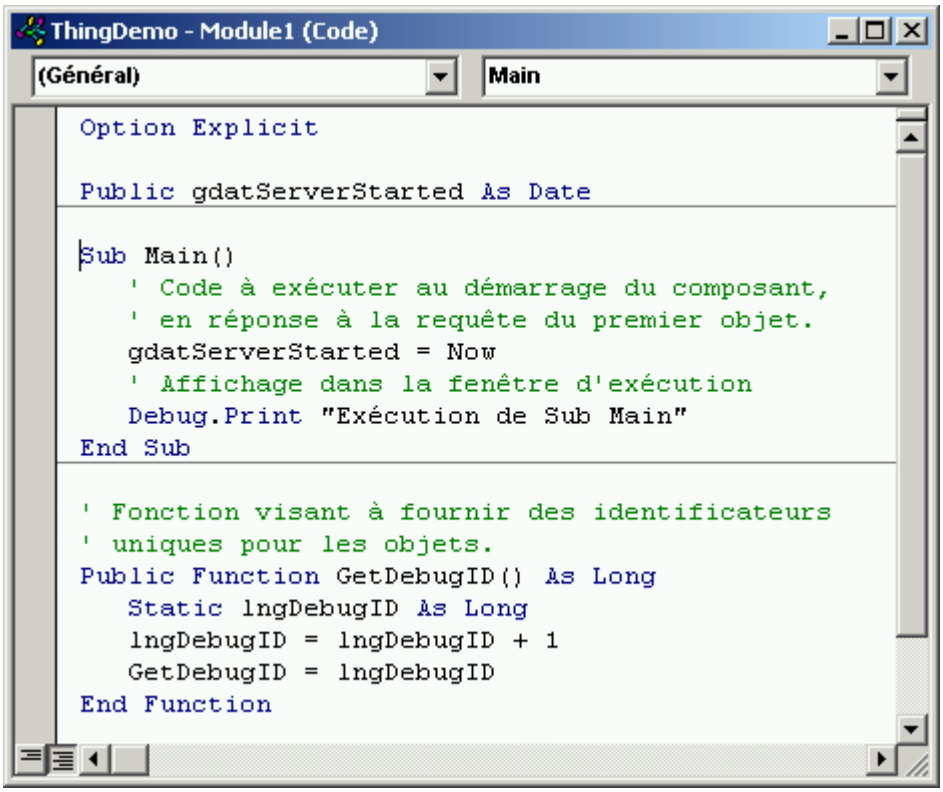
❷ Le nom du projet est utilisé comme nom de la bibliothèque de types du composant, afin d'être associé au nom de chaque classe fournie par celui-ci (produisant ainsi des noms de classes uniques).

❸ Définit le texte descriptif affiché dans le volet Description en bas de l'Explorateur d'objets ; ce texte apparaît également dans la boîte de dialogue Références.

❹ Lorsque cette case est cochée, le projet met automatiquement à jour les contrôles ActiveX à condition que vous ayez installé une nouvelle version de ces fichiers sur votre machine depuis la dernière ouverture du projet. Si une nouvelle version du contrôle ActiveX est disponible et que cette option n'est pas sélectionnée, une boîte de dialogue apparaît pour vous demander de mettre le contrôle à niveau. Cette option est sélectionnée par défaut.

Les autres onglets de cette fenêtre seront expliqués plus loin, notamment au chapitre concernant la gestion des versions des DLL.

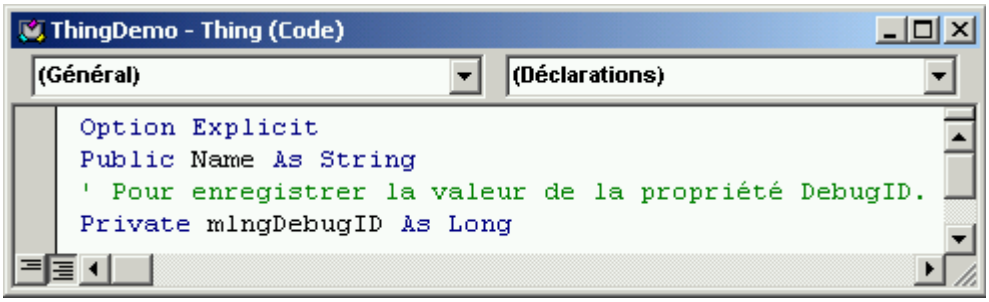
Ajoutons maintenant un module à notre projet (Projet → Ajouter un module) et glissons-y le code nécessaire à l'initialisation du composant :



Enregistrons le module sous le nom « `ThingDemo_Module1.bas` », le module de classe sous le nom « `ThingDemo_Thing.cls` », et le projet sous le nom « `ThingDemo.vbp` ».

4.2 Création des propriétés et des méthodes pour la classe `Thing`

Les propriétés d'une classe sont créées en ajoutant des variables publiques et des procédures de propriétés au module de classe. Nous allons donc créer une propriété `Name`, une chaîne de caractères qui peut être extraite et définie par des applications clientes. Ajoutons le code suivant à la section Déclarations du module de classe `Thing` :

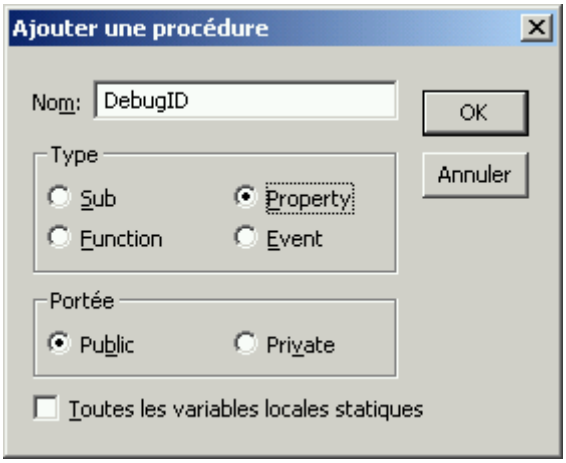


La variable `Name`[1] devient une propriété de la classe `Thing`, car elle est déclarée publique.

[1] Attention à ne pas confondre la propriété Name que nous venons de créer avec la propriété Name du module de classe, qui permet de spécifier le nom de la classe au moment de la création.

La propriété `DebugID`, en lecture seule, renvoiera un numéro de séquence indiquant l'ordre dans lequel les objets `Thing` seront créés. Nous verrons plus loin – lors du déboguage – en quoi cette information nous sera utile.

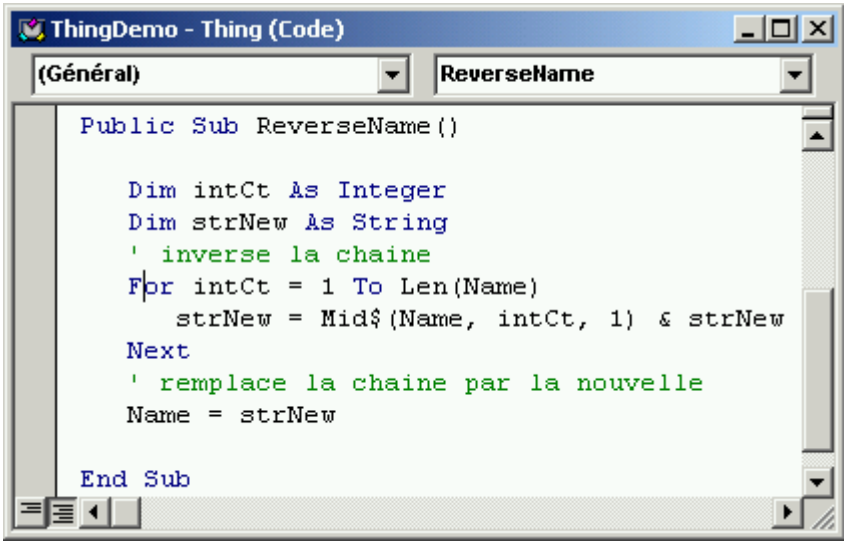
Nous allons ensuite ajouter une procédure (Outils → Ajouter une procédure…), de nom `DebugID`. Cliquons sur les boutons radio Property et Public, puis après avoir validé, nous constatons que le squelette des procédures `Get DebugID` et `Let DebugID` ont été créés.
Supprimons la procédure `Property Let`, dont le rôle est de permettre aux utilisateurs d'affecter une nouvelle valeur à la propriété `DebugID`. En la supprimant, la propriété sera donc en lecture seule. Modifions ensuite `Property Get` de manière à ce qu'elle renvoie la valeur de la variable privée, permettant ainsi aux clients de lire la valeur de la propriété :




```
Public Property Get DebugID() As Long
      DebugID = mlngDebugID
End Property
```

La variable `mlngDebugID` est un membre de données privé, utilisé pour mémoriser la valeur de la propriété `DebugID`. Étant donné que cette variable est déclarée Private, elle n'est pas visible pour les applications client qui ne peuvent donc la modifier[2].

Maintenant que nous avons vu comment créer des propriétés, voyons comment faire pour des méthodes. De la même manière que précédemment, créons une nouvelle procédure (Outils → Ajouter une procédure…) nommée `ReverseName`, de type Sub, et de portée Public. Cette méthode inversera l'ordre des lettres dans la propriété `Name`. Voyons donc le code permettant de réaliser ceci :

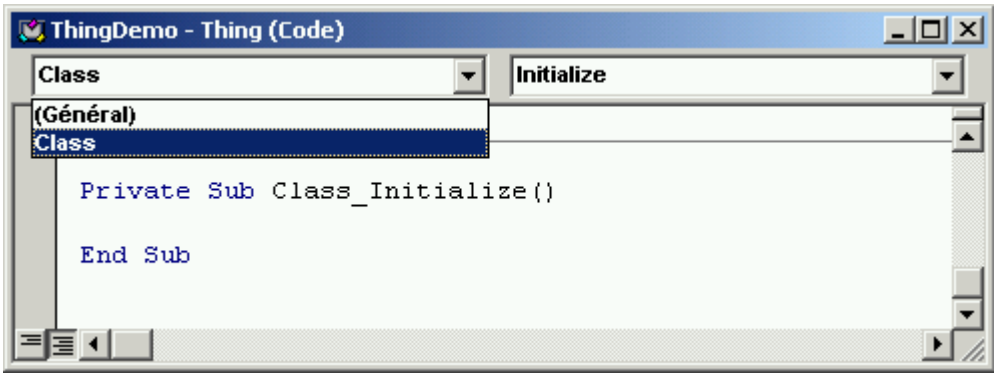


[2] Ce qui est un exemple d'encapsulation : l'objet – qui n'est ni une structure de programme, ni une structure de données – regroupe en une même entité à la fois les attributs (données) et les opérations ou méthodes (sous-programmes) associés. La structure interne (implémentation) est cachée afin de garantir l'indépendance de la vue externe (abstraite) par rapport à la vue interne.

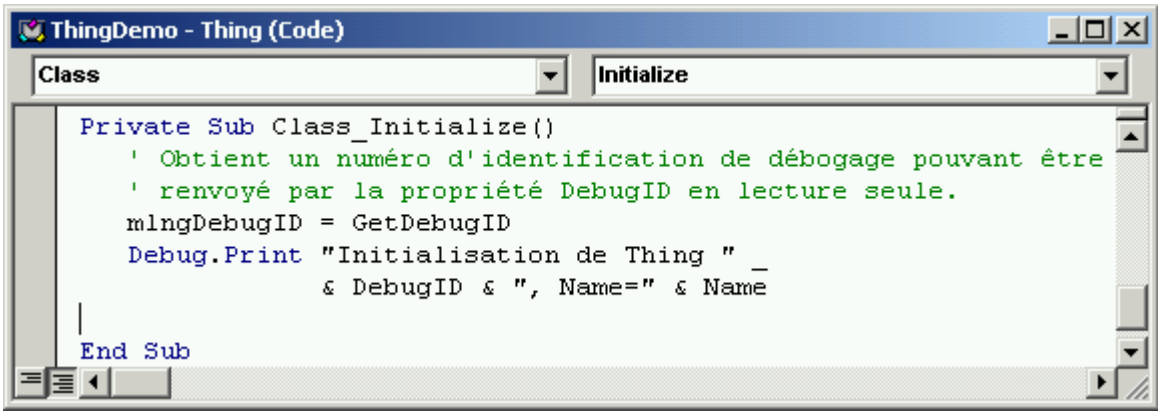
4.3 Ajout d'un code pour les événements `Initialize` et `Terminate`

Les modules de classe comportent deux événements intégrés : `Initialize` et `Terminate`. Le code que nous placerons dans la procédure événementielle `Initialize` sera le premier code exécuté quand l'objet sera créé, avant la définition de toute propriété ou l'exécution de toute méthode.
Le code que nous placerons dans l'événement `Terminate` sera exécuté quand toutes les références à l'objet auront été libérées et que l'objet sera sur le point d'être détruit.

Complétons donc la procédure d'initialisation de notre classe (en cliquant sur Class dans la zone objet du module de classe `Thing`, comme le montre la figure suivante)



Nous constatons que l'événement `Initialize` est apparu. Complétons-le ainsi :



Cette procédure permet de gérer la propriété `DebugID` ainsi que les méthodes `Debug.Print`, qui afficheront les propriétés de l'objet dans la fenêtre d'exécution au moment de sa création et de sa destruction.

De la même manière, la procédure `Terminate` (atteinte en la sélectionnant depuis le menu déroulant procédure du module de classe) est complétée afin de gérer les erreurs. Ces dernières ne pouvant être gérées par l'application utilisant le composant, elles pourraient être fatales à celle-ci si elles ne sont pas traitées :

```
Private Sub Class_Terminate()
   On Error Resume Next
   Debug.Print "Fin de Thing " & DebugID & ", Name=" & Name
End Sub
```

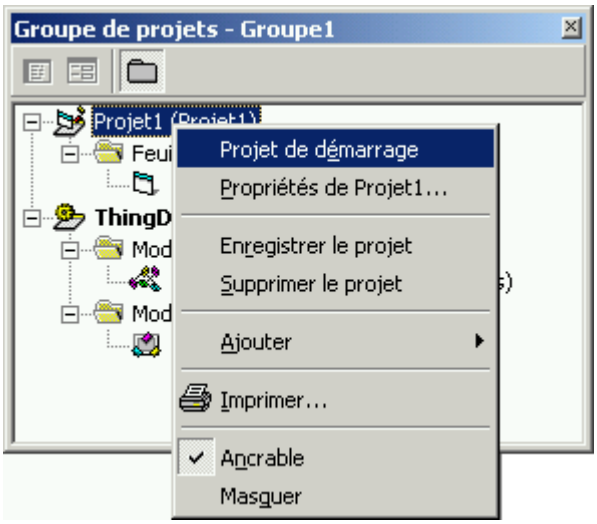
Par contre, les erreurs non gérées dans l'événement `Initialize` seront déclenchées au point où l'application a créé l'objet, et pourront donc être gérées par elle.

L'événement `Terminate` est le dernier de la vie d'un objet. On peut placer du code de nettoyage dans la procédure d'événement `Class_Terminate` afin qu'il soit exécuté au moment de la libération de la dernière référence à l'objet, juste avant la destruction de celui-ci. Les objets complexes qui contiennent des objets dépendants doivent libérer toutes les références à leurs objets dépendants dans l'événement `Terminate`.
Les erreurs qui se produisent dans l'événement `Terminate` exigent une gestion rigoureuse. En effet, l'événement `Terminate` n'étant pas appelé par l'application client, il n'est suivi d'aucune procédure dans la pile des appels. Autrement dit, une erreur non gérée dans un événement `Terminate` engendrera une erreur fatale dans le composant. Dans le cas des composants in-process (notre DLL en est un), cette erreur fatale est l'erreur fatale du client. Le composant étant exécuté dans le processus du client, l'erreur fatale d'un composant provoquera l'arrêt de l'application client.

4.4 Création du projet test TestThing

Pour tester notre composant ThingDemo, il nous faut un projet de test, qui créera des instances des classes fournies par notre composant et testera leurs propriétés, méthodes et événements. Nous utiliserons également un groupe de projets, qui nous permettra d'activer le déboguage in-process.

Le projet à ajouter s'effectue par Fichier → Ajouter un projet… où nous sélectionnons un projet de type EXE Standard.

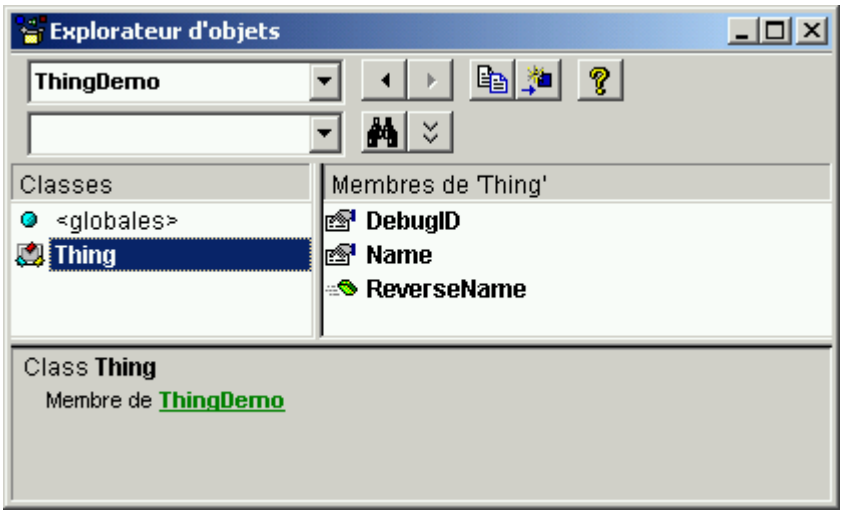


A ce niveau, le projet de démarrage (symbolisé par des caractères en gras) est ThingDemo. Etant donné que nous développons une DLL, il faut que le projet de démarrage soit l'EXE : en cliquant avec le bouton droit sur Projet1 dans la fenêtre du projet, et en sélectionnant Projet de démarrage, on assure que ce sera bien le projet de test qui s'exécutera en appuyant sur F5 (exécution).

Nous allons ensuite référencer le composant ThingDemo dans notre projet. Pour ce faire, sous Projet → Références, cochons ThingDemo puis validons[3].

[3] Si ThingDemo n'apparaît pas dans la liste des références disponibles, il faut annuler, cliquer sur Projet1 dans la fenêtre de projet pour activer celui-ci, et recommencer le référencement. Lorsqu'on travaille avec des groupes de projets, il faut toujours s'assurer que le projet actif est le bon avant de cliquer sur le menu Projet.

Si on ouvre maintenant l'explorateur d'objets (F2), et que l'on recherche l'objet Thing, on constate que l'on a à disposition les deux propriétés et la méthode déclarées auparavant.

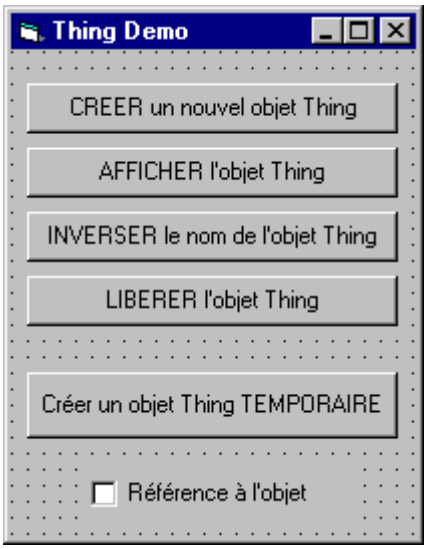


Enregistrons maintenant tous nos fichiers (Fichier → Enregistrer le groupe de projets). Nommons la feuille « ThingTest_Form1.frm », le projet test « ThingTest.vbp », et le groupe de projets « ThingDemo.vbg ».

4.5 Création et test des objets Thing

Nous allons maintenant tester notre classe Thing. Pour cela, le projet test doit demander que le composant crée un objet à partir de cette classe.

Nous utiliserons la feuille du projet test pour créer une petite interface graphique qui nous permettra de créer et de détruire ces objets. Voilà à quoi ressemblera cette fenêtre (cinq boutons Command et une case à cocher) :



Complétons maintenant le code de manière à faire une référence à l'objet Thing depuis la feuille de l'application test (dans la section Déclarations) :

```
' Référence à un objet Thing.
Private mthTest As Thing
```

Ajoutons ensuite le code des procédures événementielles Click des boutons 1 à 4.

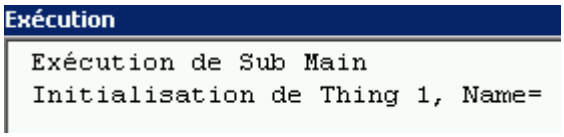
```
Option Explicit
' Référence à un objet Thing.
Private mthTest As Thing
' Bouton "CREER un nouvel objet Thing".
Private Sub Command1_Click()
   Set mthTest = New Thing
   mthTest.Name = InputBox("Tapez un nom pour l'objet Thing", "Test Thing")
End Sub
' Bouton "AFFICHER l'objet Thing".
Private Sub Command2_Click()
   MsgBox "Name: " & mthTest.Name, , "Thing " & mthTest.DebugID
End Sub
' Bouton "INVERSER le nom de l'objet Thing".
Private Sub Command3_Click()
   mthTest.ReverseName
   ' Clic sur "AFFICHER l'objet Thing" en définissant sa valeur.
   Command2.Value = True
End Sub

' Bouton "LIBERER l'objet Thing".
Private Sub Command4_Click()
   Set mthTest = Nothing
End Sub
```

Nous ne nous occuperons pas pour l'instant du cinquième bouton ni de la case à cocher.

4.6 Premier test : exécution de l'application TestThing

Nous allons maintenant nous pencher sur les principaux événements intervenant pendant la durée de vie d'un objet fourni par un composant in-process, y compris ce qui se passe lorsque la DLL est déchargée de force en cliquant sur le bouton Fin. Les messages qui nous intéresseront apparaîtrons dans la fenêtre d'exécution (Ctrl+G).

Démarrons donc l'exécution (Ctrl+F5)[4].
Cliquons sur « CREER un nouvel objet Thing », et avant de compléter un nom pour le nouvel objet, observons la fenêtre d'exécution :



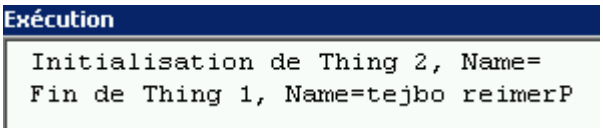
[4] Par défaut, l'option Compilation sur demande est cochée dans l'onglet Général de la boîte de dialogue Options (accessible depuis le menu Outils). Si, lors du déboguage d'un composant, l'option Compilation sur demande est cochée, vous pouvez utiliser la combinaison de touches Ctrl+F5 (ou Exécuter avec compilation complète, du menu Exécution) pour compiler tous les projets du groupe avant de lancer le mode Exécution. Les erreurs de compilation requièrent la réinitialisation du projet, ce qui signifie un retour en mode Création.

Quand la procédure événementielle Command1_Click crée l'objet Thing test, deux choses se produisent. Premièrement, avant que l'objet soit créé, le code contenu dans `Sub Main` est exécuté. C'est seulement après cette exécution que l'objet Thing est créé[5]. Notons que la propriété Name n'a pas encore de valeur.

Saisissons « Premier objet » dans la boîte apparue et validons. Cette valeur est alors affectée à la propriété Name de l'objet `Thing`. Cliquons maintenant sur « AFFICHER l'objet Thing » : la propriété `Name` possède la valeur que nous venons de lui donner.

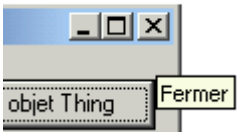
Cliquons ensuite sur « INVERSER le nom de l'objet Thing » afin d'appeler la méthode `ReverseName`. Le nom de l'objet apparaît inversé, selon le code que nous avons inséré dans la procédure `Command3_Click()`.

Cliquons à nouveau sur « CREER un nouvel objet Thing », et observons le contenu de la fenêtre d'exécution :



un nouveau message de déboguage s'affiche : c'est celui de l'événement `Initialize` du nouvel objet créé, alors que l'opérateur `New` provoque la création de l'objet. Le second message émane de l'événement `Terminate` du premier objet `Thing` que nous avons créé. Celui-ci est créé lorsque la référence au nouvel objet est placée dans la variable `mthTest` – à ce moment, aucune variable ne contient les références au premier objet `Thing`, qui doit donc être détruit.

Saisissons « Deuxième objet », validons, et fermons l'application en cliquant sur le bouton fermer, et non sur le bouton Fin de VB.



Avant la fermeture du programme, le message `Terminate` pour l'objet que nous venons de créer (le second) s'affiche dans la fenêtre d'exécution. Lorsqu'on ferme un programme comme on l'a fait, VB élimine toutes les variables qui contiennent encore des références d'objet[6]. Dans la mesure où toutes les valeurs de toutes les variables sont définies sur `Nothing`, l'objet auquel elles font référence doit être détruit.

Lorsqu'un programme est arrêté brusquement avec le bouton Fin de la barre d'outils de VB, ou alors avec une instruction `End` dans le code, VB récupère la mémoire et les ressources que le programme utilisait. Malgré cela, l'élimination des données se produit comme si le programme avait subi une erreur fatale, et les objets n'obtiendront jamais leurs événements `Terminate`.

---

[5] La procédure Sub Main d'un composant ActiveX est exécutée quand le composant reçoit la première requête pour l'un des objets qu'il fournit, avant que le composant crée l'objet. Il convient de ne pas intégrer des tâches longues dans la procédure Sub Main, car une requête de création d'un objet risque d'expirer en attendant l'exécution de Sub Main.

[6] La principale règle régissant la durée de vie d'un objet est très simple : un objet est détruit lorsque la dernière référence le désignant est libérée. Pour optimiser l'utilisation des objets, COM définit un ensemble de raccourcis complexe dans les règles de dénombrement des références. Il nous serait donc impossible de nous fier à ce nombre de références, si nous y avions accès.
Selon les règles de COM, la seule information fiable indique si le *nombre de références est égal à zéro*. Nous sommes informé que le nombre de références a atteint zéro, car l'événement `Terminate` de notre objet se produit. Ceci mis à part, le nombre de références ne nous apporte aucune information fiable.

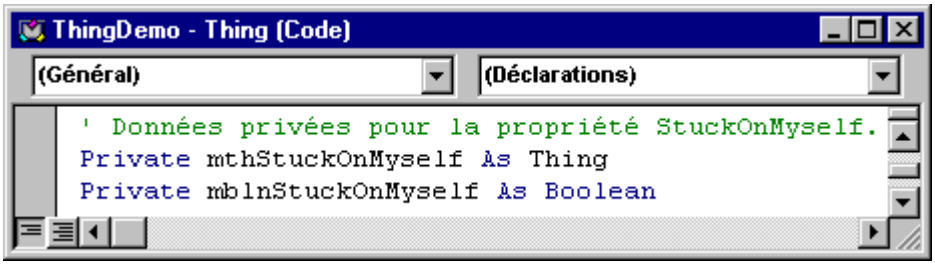
Pour s'en convaincre, il suffit de relancer notre petit programme, de créer un objet Thing, de lui donner un nom, de valider, puis de quitter l'exécution en cliquant sur le bouton Fin de la barre d'outils de VB : Le message Terminate pour notre objet ne s'affiche pas dans la fenêtre d'exécution.

4.7 Références circulaires et durée de vie des objets

Comme nous venons de le voir, lorsque le programme fonctionne normalement, un objet n'est pas détruit avant que toutes ses références n'aient été libérées. Ce phénomène a des implications sur la durée de vie de l'objet. Les procédures de ce chapitre illustrent ce fait en accordant aux objets le privilège de s'adonner à une petite dose de narcissisme.

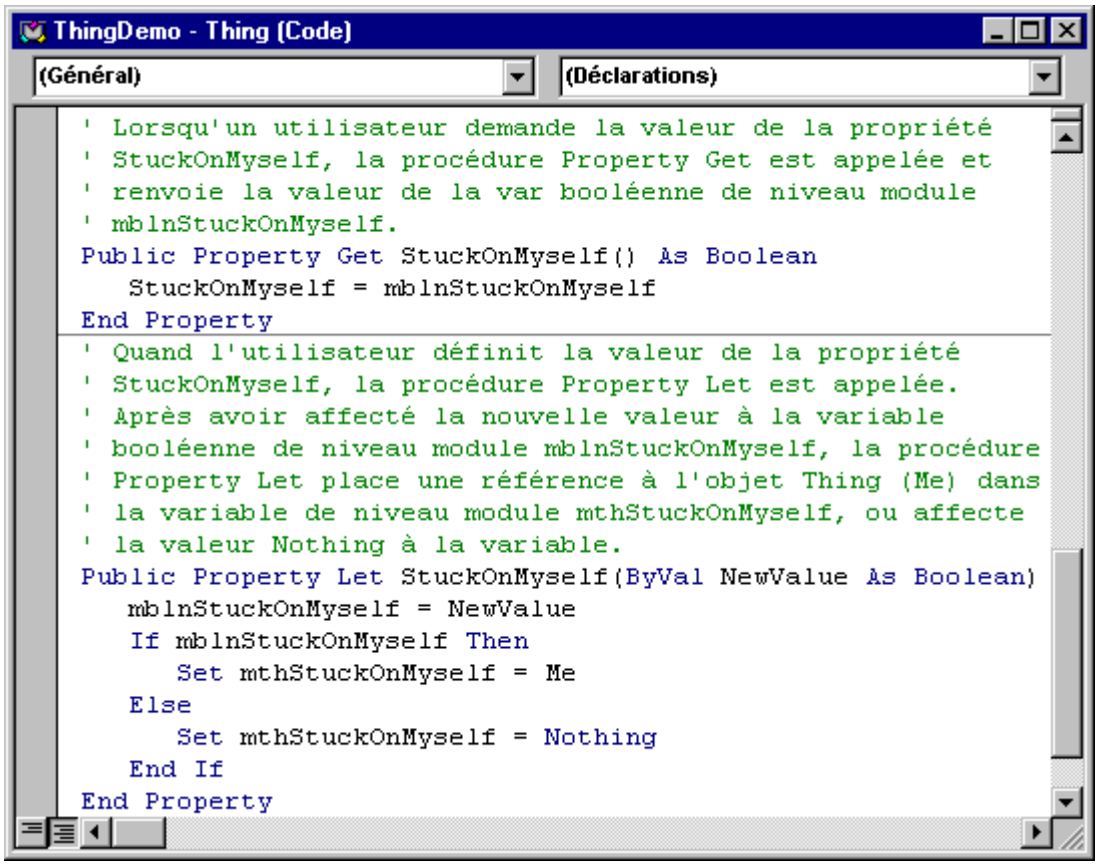
Nous utiliserons des procédures de propriétés[7] associées au code du dernier bouton de notre feuille, ainsi qu'au bouton check. Ajoutons donc la propriété `StuckOnMyself` dans la classe `Thing`.

Ainsi, ajoutons le code suivant dans la section Déclarations du module de classe `Thing` :



[7] En règle générale, les propriétés des objets fournis par les composants doivent être implémentées en tant que procédures de propriétés. Celles-ci sont plus fiables que les membres de données. Une propriété de type énumération, par exemple, ne peut être validée que si elle est implémentée sous forme de procédures `Property Get` et `Property Let`.
Cette règle connaît une seule exception : une propriété de chaîne ou numérique simple qui ne nécessite aucune validation et dont la modification n'affecte pas immédiatement les autres propriétés de l'objet.
Une propriété objet (autrement dit, une propriété qui contient une référence d'objet plutôt qu'un type de données ordinaire) doit presque toujours être implémentée avec des procédures de propriétés. Une propriété objet implémentée en tant que variable objet publique peut se voir affecter fortuitement la valeur `Nothing`, ce qui peut entraîner la destruction de l'objet.

Puis insérons une procédure (Outils → Ajouter une procédure…) de nom `StuckOnMyself`, de type Propriété, et de portée Publique, et modifions son code de la manière suivante :

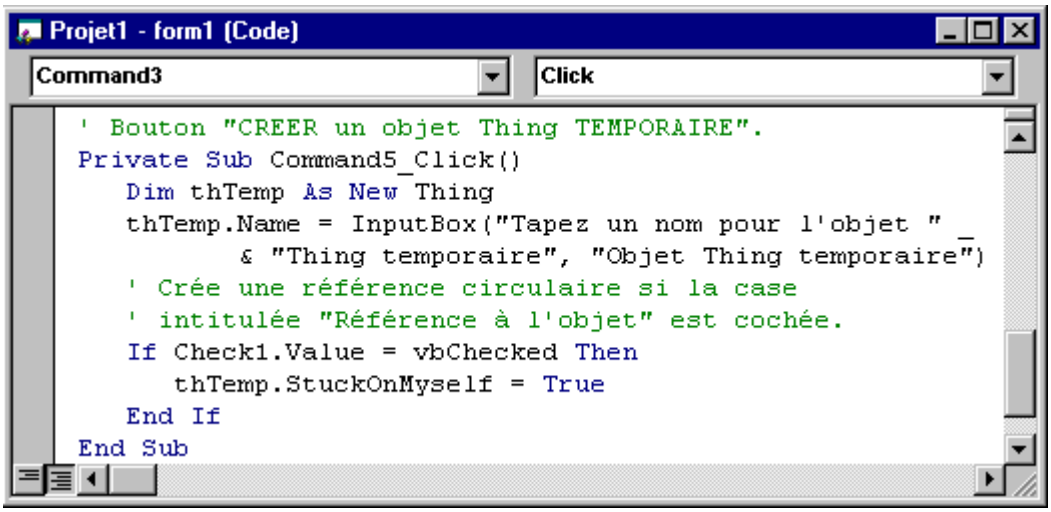


Les références circulaires : qu'est-ce donc ?

Une référence circulaire[8] se produit quand deux objets contiennent des références l'un par rapport à l'autre, ou quand un objet contient une référence à lui-même, comme c'est le cas dans la propriété `StuckOnMyself`.

Nous allons ajouter du code dans le bouton « CREER un objet Thing TEMPORAIRE » afin de démontrer les références circulaires en testant de manière sélective la propriété `StuckOnMyself`.

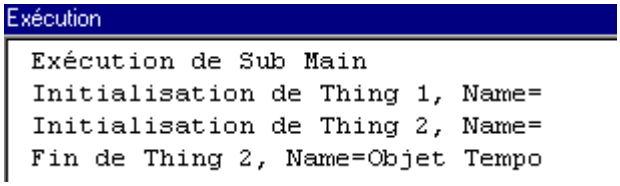
[8] Visual Basic détruit un objet lorsqu'il n'existe plus aucune référence à celui-ci. Si le parent d'un objet possède une collection qui renferme une référence à l'objet, cela suffit à empêcher la destruction de l'objet. De même, un objet continue d'exister aussi longtemps que le parent possède une propriété d'objet qui contient un référence à l'objet.
Lorsqu'un objet parent est détruit, les variables qui implémentent ses propriétés se situent hors portée et les références à l'objet sont libérées. Cela permet de fermer les objets dépendants. Toutefois, si un objet dépendant possède une propriété Parent, Visual Basic est incapable de détruire l'objet parent, étant donné que l'objet dépendant renferme une référence à celui-ci.
Étant donné qu'un composant in-process partage l'espace de processus de l'application client, il n'existe aucune distinction entre des références « externes » et « internes » à un objet public. Le composant demeure chargé aussi longtemps qu'il existe une référence à l'un des objets qu'il fournit.
Cela signifie qu'une référence circulaire prolonge indéfiniment le chargement d'un composant in-process et que la mémoire occupée par les objets orphelins ne peut être libérée aussi longtemps que l'application client n'est pas fermée.



Notons qu'aucun code n'est requis pour la case à cocher. Si cette dernière est cochée, la procédure d'événement `Command5_Click` définit la propriété `StuckOnMyself` du nouvel objet `Thing`.
Au lieu de créer explicitement un nouvel objet `Thing` avec l'opérateur `New`, le bouton Objet Thing temporaire utilise une variable déclarée `As New`, permettant la création implicite de l'objet.
La procédure suivante fournit la démonstration correspondante. Elle montre aussi de quelle manière la durée de vie de l'objet est affectée par la portée de la variable et illustre les effets des références circulaires sur la durée de vie de l'objet.

Voyons donc comment fonctionnent les références circulaires. Exécutons le groupe de projets (Ctrl+F5), et créons un nouvel objet nommé « Objet à long terme », et validons.
A présent, cliquons sur « Créer un objet Thing TEMPORAIRE ». Puisque la variable objet `thTemp` qui contient la référence à cet objet `Thing` est une variable de niveau procédure, sa durée de vie - et par conséquence la durée de vie de l'objet - se limite à l'exécution de la procédure.
La fonction InputBox apparaît : Visual Basic doit évaluer le code placé à droite du signe égal avant d'affecter le résultat à la propriété `Name` du nouvel objet `Thing`.
Avant de taper un nom dans la zone d'entrée, observons la fenêtre d'exécution. Elle ne contient aucun message `Initialize` de la part du nouvel objet `Thing`, puisqu'il n'a pas encore été créé. Dans la mesure où la variable `thTemp` a été déclarée `As New`, un objet `Thing` ne sera créé que lorsque l'une de ses propriétés ou de ses méthodes sera appelée, pas avant.

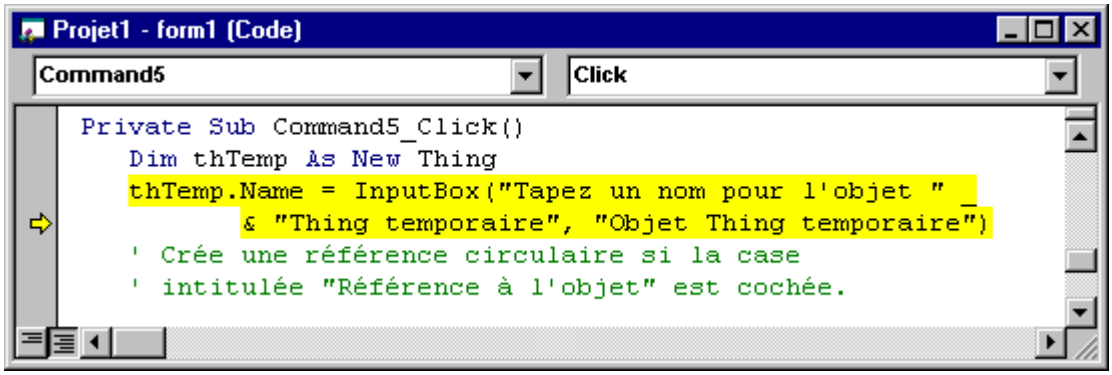
Tapons « Objet Tempo » dans l'InputBox, et validons.
Deux messages s'affichent dans la fenêtre Exécution, un message `Initialize` et un message `Terminate`. L'événement `Initialize` se produit au moment où le nom indiqué dans InputBox est affecté à `thTemp.Name`. Visual Basic trouve que `thTemp` contient `Nothing`, crée un objet `Thing` et place une référence à l'objet dans `thTemp`.



Bien que la valeur de la propriété `DebugID` ait déjà été définie — c'est ce qui se passe en premier dans l'événement `Initialize` — la propriété `Name` est toujours vide. Cet état de chose souligne le fait que l'événement `Initialize` se produit avant l'exécution de tout autre code ou la définition de toute propriété.
C'est uniquement après, que Visual Basic est en mesure d'affecter la valeur de la fonction InputBox à la propriété `Name` de l'objet `Thing`. Beaucoup de travail pour une seule ligne de code !

Dès que l'objet `Thing` est créé, la procédure d'événement `Command5_Click` se termine. La variable `thTemp` devient hors de portée, exactement comme si sa valeur `Nothing` lui avait été affectée. Il n'existe plus aucune référence à l'objet `Thing` temporaire, il est donc détruit. Son événement `Terminate` affiche ses propriétés, y compris le nom que vous lui avait attribué.

Passons maintenant en mode arrêt (Ctrl+Pause), puis passons en mode pas à pas (F8). Créons un nouvel objet temporaire afin d'entrer dans la procédure événementielle `Command5_Click`, et appuyons à nouveau sur F8 pour avancer sur la ligne définissant le nom de l'objet `Thing`.



Une nouvelle fois F8, et nous arrivons sur la ligne de code exécutant l'instruction InputBox. Tapons « second tempo » et validons.

Puisque nous déboguons le composant `ThingDemo` dans le même environnement que le programme test, Visual Basic peut entrer directement dans le code du composant à partir du programme test.
Continuons d'appuyer sur F8 pour afficher le code correspondant à l'événement `Initialize`, à la propriété `DebugID` et à l'événement `Terminate`. Quand nous arrivons à la dernière ligne de `Class_Terminate()`, appuyons sur F5 pour revenir au mode Exécution.

Le débogage in-process est un outil puissant qui permet de connaître l'ordre dans lequel les événements se produisent dans un composant.

Cochons maintenant « Référence à l'objet », puis créons à nouveau un objet temporaire (qui comportera une référence à lui-même), et appelons le « Renégat »…
A la validation de l'InputBox, Dans la fenêtre d'exécution s'affiche un message `Initialize` pour l'objet `Thing`, mais pas de message `Terminate`. L'objet n'a pas été détruit quand la variable `thTemp` est devenue hors de portée, car il existait encore une référence à l'objet dans la variable `mthStuckOnMyself` appartenant à la propriété `StuckOnMyself`.

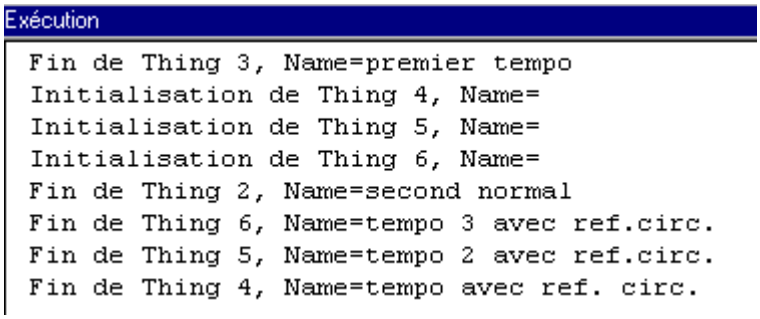
Quand cet objet `Thing` temporaire sera-t-il donc détruit ? Si on pouvait affecter la valeur `False` à la propriété `StuckOnMyself` de l'objet `Thing` Renégat, il n'existerait plus aucune référence, mais `TestThing` ne peut effectuer cette opération, puisque le programme ne possède plus une référence à utiliser pour appeler la propriété `StuckOnMyself` !

L'objet est devenu orphelin et il continuera à utiliser de la mémoire jusqu'à ce que la DLL soit déchargée. Comme nous le verrons dans une procédure ultérieure, la référence circulaire dans la propriété `StuckOnMyself` de l'objet `Thing` Renégat conservera la DLL complète en mémoire.

Cliquons sur « Libérer l'objet Thing » pour détruire l'objet `Thing` que nous avions appelé « Objet à long terme », en observant son message `Terminate` dans la fenêtre d'exécution, puis créons plusieurs fois des objets temporaires, en leur donnant à chacun un nom différent.

Comme pour le premier objet Renégat, il n'y aura pas de message `Terminate` pour ces objets dans la fenêtre d'exécution, car les références circulaires empêcheront leur destruction.

Et finalement, fermons le test (clic dans la zone Fermer, pas sur le bouton Arrêt !) pour revenir en mode création.



Abracadabra ! Comme par magie, toute une série de messages `Terminate` apparaissent dans la fenêtre d'exécution.

Quand Visual Basic arrête `TestThing`, il arrête aussi `ThingDemo`. De nature ordonnée, il efface toutes les variables objets de `ThingDemo`, y compris les références associées contenues dans l'objet `Thing` Renégat et dans sa suite.

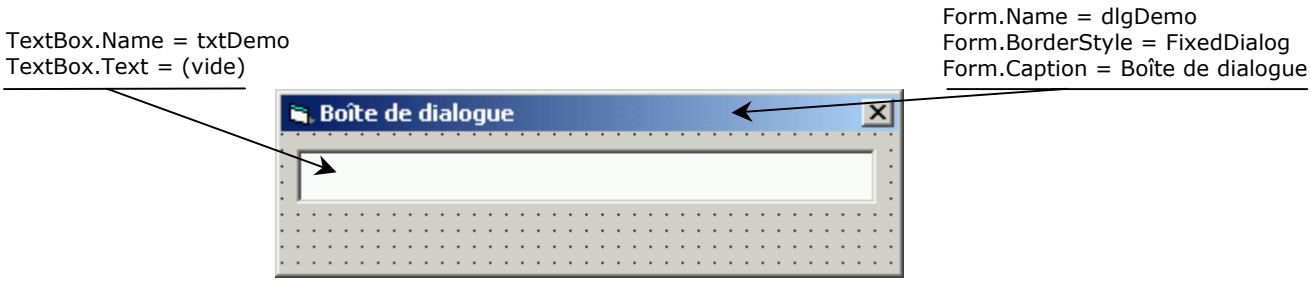
Comme on le constate, il est important d'éviter de conserver des références d'objets superflues dans un composant in-process. Un client peut créer et libérer des centaines d'objets pendant qu'il utilise un composant. Si tous ces objets sont conservés en mémoire, à l'instar des objets `Thing` Renégat, les performances finiront par se détériorer.

4.8 Ajout d'une feuille au projet ThingDemo

Les composants in-process peuvent être utilisés comme bibliothèques de procédures et de boîtes de dialogue. Ils permettent ainsi d'économiser du temps de programmation tout en donnant aux applications une présentation homogène et cohérente.

Les procédures de ce chapitre montrent la manière dont peuvent être utilisés les objets pour contrôler des boîtes de dialogue modales ou non modales[9].

La même feuille sera utilisée dans les deux cas.



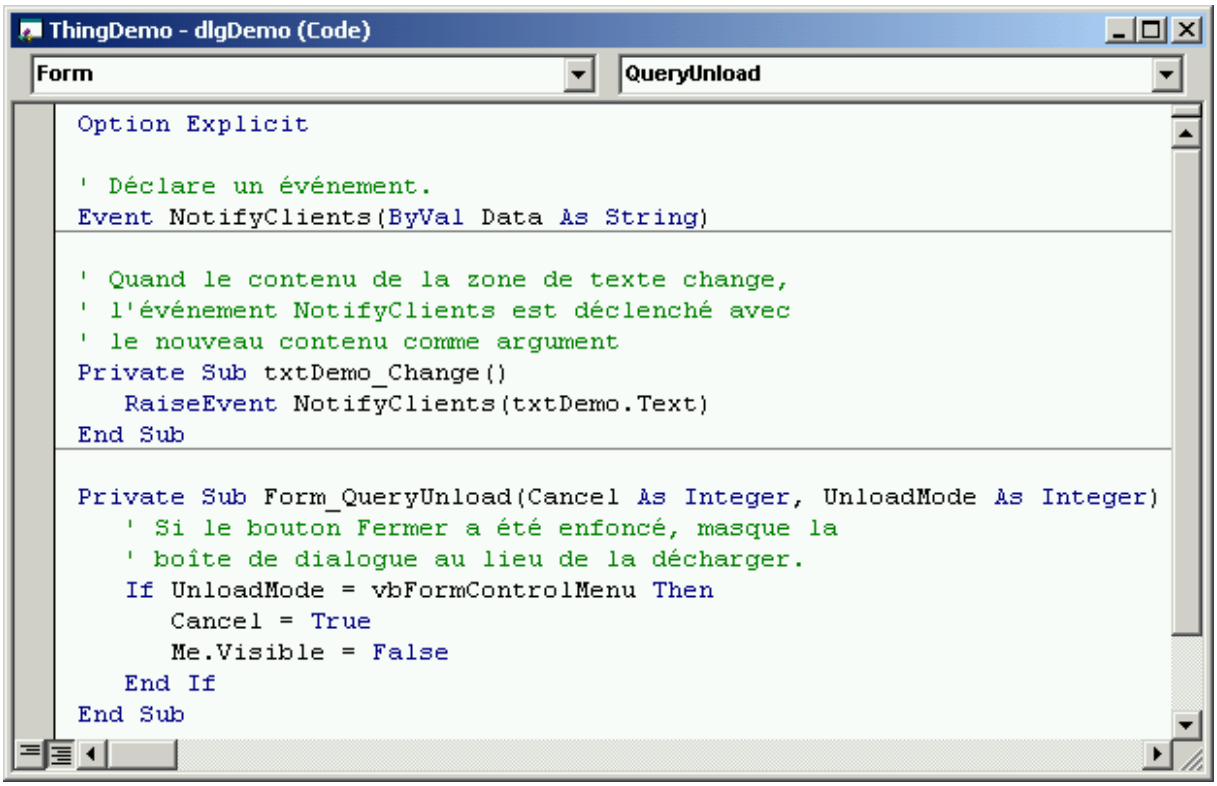
Pour ajouter cette feuille, il faut d'abord sélectionner `ThingDemo` – pour en faire le projet actif – et sélectionner dans le menu Projet l'article Ajouter une feuille. L'édition de celle-ci s'effectue comme pour n'importe quelle autre feuille. Enregistrons-la sous le nom `ThingDemo_dlgDemo.frm`.

Notons que la boîte de dialogue ne sera pas appelée directement par le client, car les feuilles sont des classes privées. Les clients ne peuvent créer des instances de classes privées et on ne doit jamais transmettre des instances de classes privées aux applications client[10].
Pour afficher la boîte de dialogue, les clients appelleront la méthode `ShowDialog` d'un objet `Dialogs` global qui créera et affichera la feuille `dlgDemo`.

[9] On distingue des boîtes de dialogue modales et non modales. Une boîte de dialogue modale doit être fermée (masquée ou déchargée) pour pouvoir poursuivre le travail. Par exemple, une boîte de dialogue est modale si elle demande de cliquer sur OK ou Annuler avant de passer à une autre feuille ou boîte de dialogue.
Dans Visual Basic, la boîte de dialogue À propos de est modale. Les boîtes de dialogue qui affichent des messages importants doivent toujours être de ce type : avant de poursuivre son travail, l'utilisateur doit toujours intervenir, soit en les fermant, soit en répondant à un message.
Les boîtes de dialogue non modales permettent de déplacer le focus entre la boîte de dialogue et une autre feuille sans avoir à fermer la boîte de dialogue. Il est possible de continuer à travailler dans l'application en cours pendant que cette boîte de dialogue est affichée. Les boîtes de dialogue non modales sont assez rares. La boîte de dialogue Rechercher du menu Edition de Visual Basic en est un exemple. Ce type de boîte de dialogue est à utiliser pour afficher des commandes fréquemment utilisées ou des informations.

[10] Les types de données suivants ne sont pas autorisés et les références à ceux-ci ne doivent jamais être renvoyées aux applications client : tous les objets fournis dans la bibliothèque d'objets Visual Basic (VB), notamment les contrôles; toutes les feuilles; tous les modules de classe dont la propriété Instancing a pour valeur Private ; les références aux contrôles ActiveX.

Ajoutons donc le code suivant à la feuille nouvellement créée :

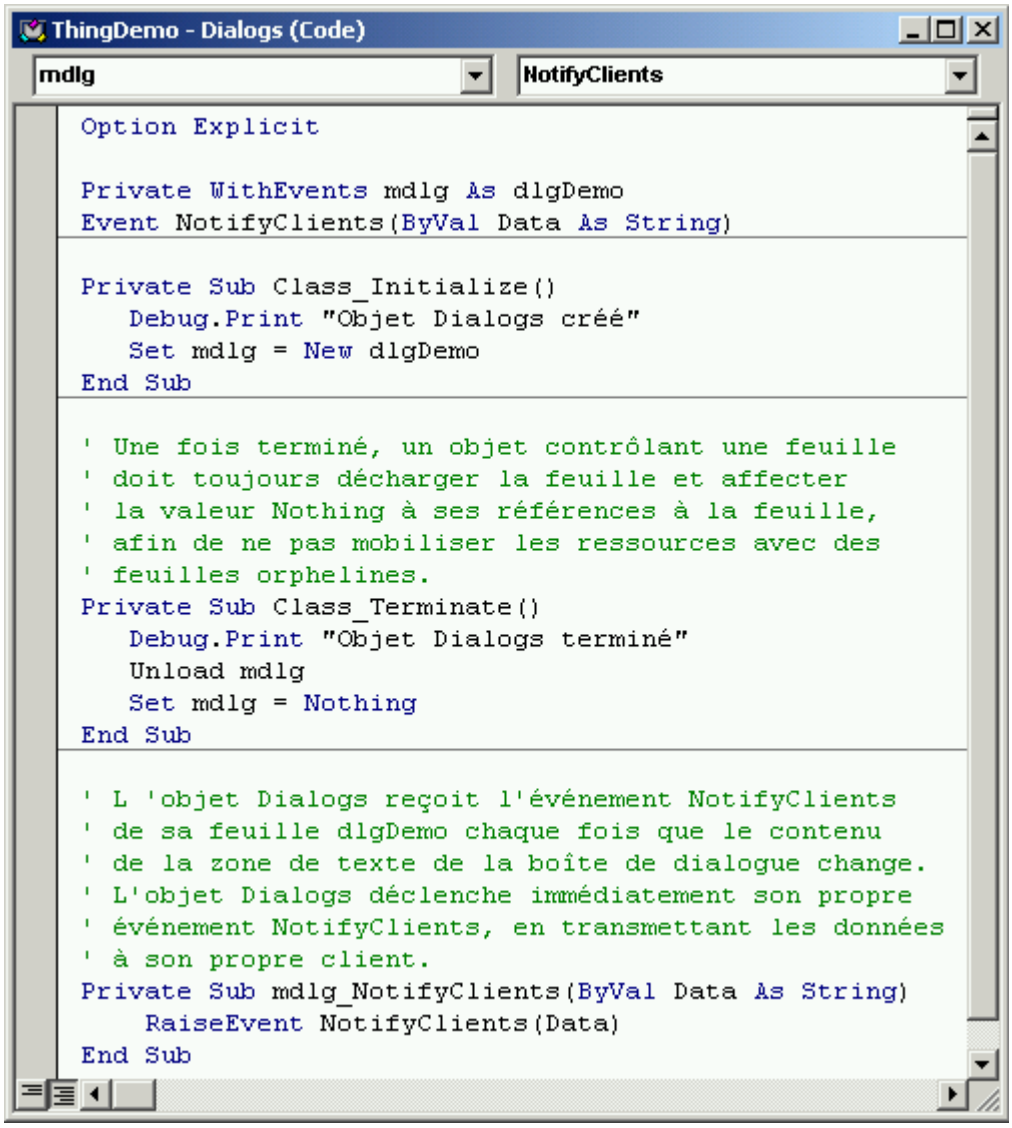


Si la feuille `dlgDemo` est décrite comme une boîte de dialogue modale, le fait de masquer la boîte de dialogue plutôt que de la décharger permet à la méthode `ShowDialog` de la classe `Dialogs` d'extraire la valeur de la zone de texte.

Créons maintenant la classe `Dialogs` (Projet → Ajouter un module de classe), et appelons-le `Dialogs` (dans sa fenêtre Propriétés). Afin de pouvoir appeler la méthode `ShowDialogs` sans créer explicitement un objet `Dialogs`, modifions sa propriété `Instancing` en `GlobalMultiUse`.

Créons ensuite une variable `WithEvents` qui puisse gérer l'événement `NotifyClients` de `dlgDemo`, ainsi qu'un événement que l'objet `Dialogs` puisse déclencher pour ses propres clients.

N'oublions pas non plus de gérer les procédures d'initialisation et de terminaison de la classe :



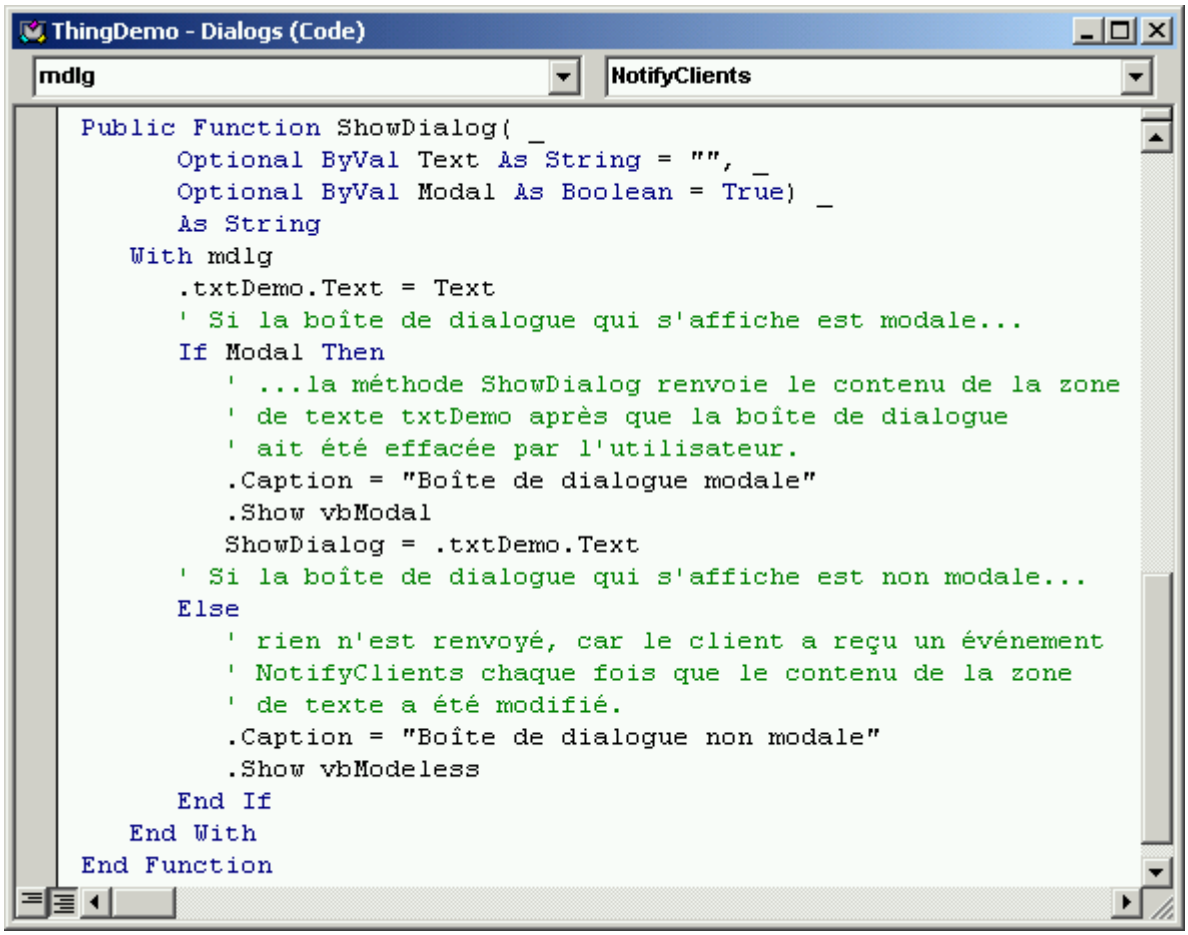
Créons la procédure – de nom `ShowDialog`, de type fonction et de portée publique - qui gérera le contenu de la boîte de dialogue, selon qu'elle sera modale ou non (Outils → Ajouter une procédure).

Modifions-la comme le montre la figure suivante. Nous utiliserons des arguments optionnels, qui permettent au compilateur de repérer les erreurs d'incompatibilité de type, au lieu d'attendre que des erreurs d'exécution se produisent.
La méthode `ShowDialog` qui affiche la boîte de dialogue comporte deux arguments optionnels :

- le texte initial qu'elle va afficher ;
- un argument booléen qui détermine si la boîte de dialogue est modale.

L'argument `Text` est affecté à la zone de texte sur `dlgDemo` avant d'afficher la boîte de dialogue. La valeur par défaut de l'argument `Modal` étant `True`, si nous omettons cet argument, la boîte de dialogue sera modale.
Si la boîte de dialogue qui s'affiche est modale, la méthode `ShowDialog` renvoie le contenu de la zone de texte `txtDemo` après que la boîte de dialogue ait été effacée par l'utilisateur.

Si la boîte de dialogue qui s'affiche est non modale, rien n'est renvoyé, car le client a reçu un événement `NotifyClients` chaque fois que le contenu de la zone de texte a été modifié.

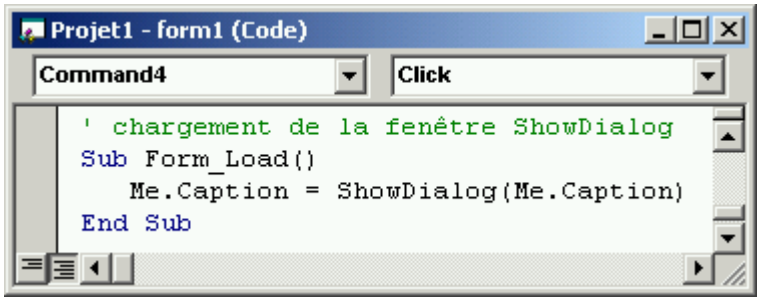


Et finalement, enregistrons notre module de classe sous le nom « `ThingDemo_Dialogs.cls` ».

4.9 Utilisation de l'objet global[11] dans TestThing

`TestThing` appellera la méthode `ShowDialog` pour afficher une boîte de dialogue modale dans sa procédure d'événement `Form_Load`, au démarrage de l'application. Elle affichera également une boîte de dialogue non modale lorsque l'on clique sur la feuille principale.

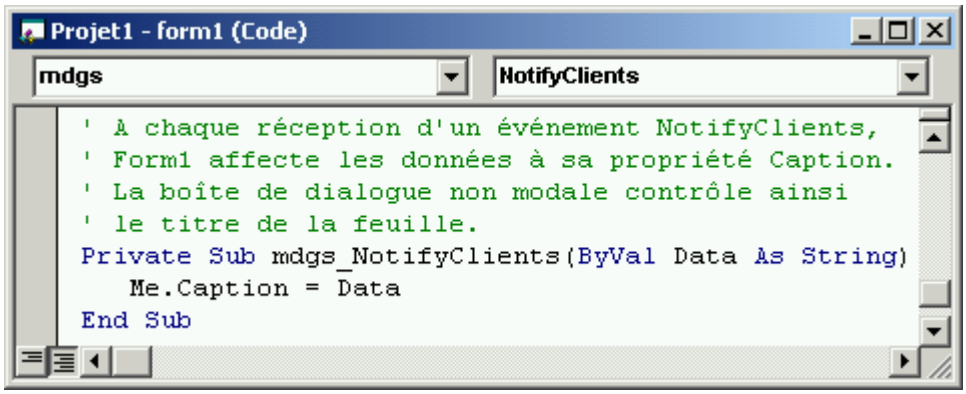
Ajoutons donc le code suivant dans l'événement `Form_Load` de la feuille Form1 :



Notons qu'aucune variable objet n'est déclarée et que la méthode `ShowDialog` est appelée comme s'il s'agissait d'une procédure `Function` ordinaire. Ceci est possible car la valeur de la propriété `Instancing` de l'objet `Dialogs` est définie sur `GlobalMultiUse`.

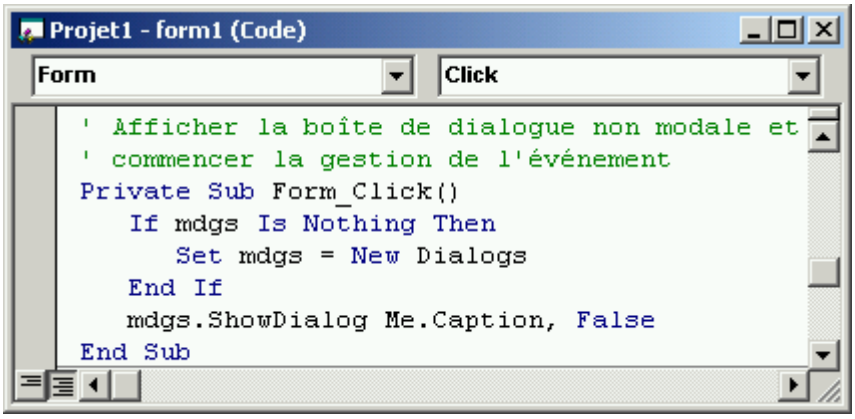
Dans la section Déclarations, ajoutons la variable `mdgs` , déclarée `WithEvents`, de sorte que `Form1` pourra gérer les événements `NotifyClients` déclenchés par l'objet `Dialogs` :

Dans la zone Objet, cliquons sur `mdgs` pour afficher la procédure d'événement et ajoutons le code suivant :



[11] Le terme « global » utilisé dans l'expression « objet global » signifie simplement que les méthodes et les propriétés de l'objet sont ajoutées à l'espace de nom global de notre projet, de manière à ce qu'elles puissent être utilisées sans déclarer au préalable une variable objet. Cela ne signifie pas qu'il n'existe qu'un seul objet de ce type, ou que plusieurs applications client peuvent partager un objet unique. Une instance de la classe sera créée pour chaque client utilisant des méthodes de la classe sans les qualifier. Un seul objet global de ce type sera créé pour chaque client.

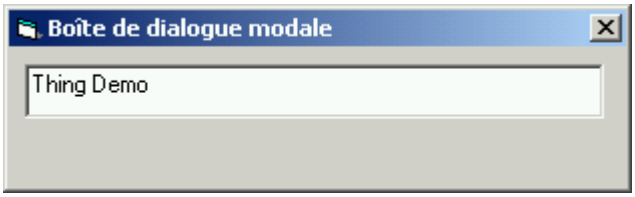
Pour afficher la boîte de dialogue non modale et commencer la gestion de l'événement, ajoutons le code suivant à la procédure d'événement `Form_Click`[12] :

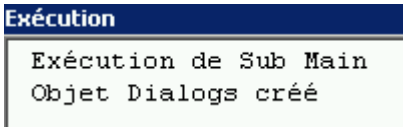


En résumé, les objets `Dialogs` créent et manipulent un objet `dlgForm` et reçoivent son événement `NotifyClients`. `Form1` créée deux objets `Dialogs` différents, l'un implicitement (l'instance globale utilisée pour afficher la boîte de dialogue modale) et l'autre explicitement (utilisé pour afficher la boîte de dialogue non modale).
La boîte de dialogue non modale communique avec l'objet `Dialogs` en déclenchant un événement `NotifyClients`. L'objet `Dialogs` répond en déclenchant son propre événement `NotifyClients` que `Form1` gère en affectant les données à sa propriété `Caption`.

Effectuons maintenant quelques test pour vérifier que tout fonctionne correctement (Ctrl+F5).

La boîte de dialogue modale contenant le titre de `Form1` s'affiche, notons également la fenêtre d'exécution :

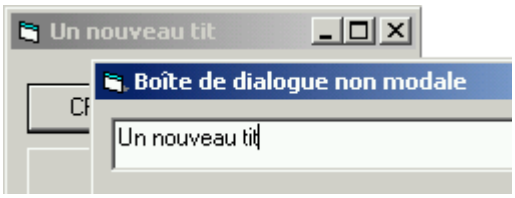




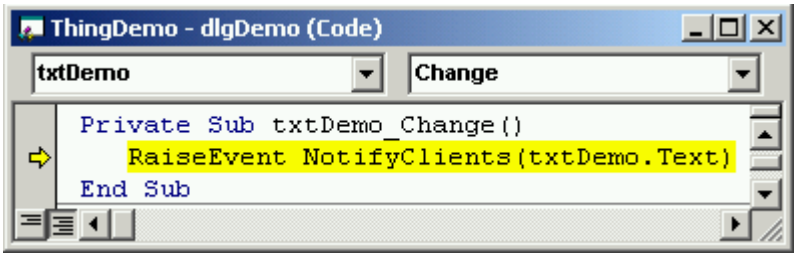
Tapons un nouveau titre (Mon titre, par exemple), et fermons la boîte de dialogue : La feuille principale s'affiche, avec le titre saisi.

[12] L'objet `Dialogs` créé ici n'est pas une instance globale utilisée pour afficher la boîte de dialogue modale dans l'événement `Load`. L'instance globale sera créée automatiquement quand la méthode `ShowDialog` sera appelée dans `Form_Load`, et utilisée pour tous les appels ultérieurs de la méthode omettant la variable objet. L'instance utilisée pour afficher la boîte de dialogue non modale est explicitement créée et affectée à la variable `WithEvents`, de manière à pouvoir gérer son événement `NotifyClients`.

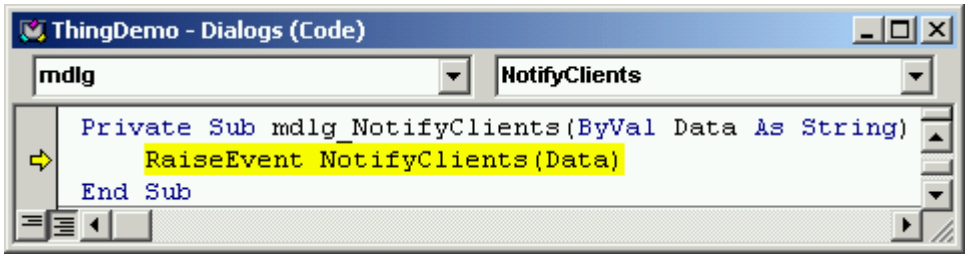
Cliquons sur la feuille principale (n'importe où, sauf sur les boutons) pour créer un autre objet `Dialogs` qui affichera un boîte de dialogue non modale cette fois contenant le titre en cours de la feuille principale. Tapons un autre titre : nous observons qu'à chaque frappe de touches, le titre de la feuille principale change.



Chaque changement apporté au contenu de la zone de texte déclenche un événement NotifyClients émanant de dlgDemo de l'objet Dialogs.



L'objet Dialogs déclenche, à son tour, son propre événement NotifyClient qui est reçu par la feuille principale.



4.10 Compilation et test de la DLL ThingDemo

Notre projet de composant in-process a été testé et débogué dans l'environnement de développement : nous pouvons le compiler et tester le fichier .dll.

Un composant ActiveX se compile de la même façon que n'importe quel projet Visual Basic, en cliquant sur l'option Créer du menu Fichier. Le fichier .dll inclut une bibliothèque de types qui décrit nos objets et nous permet de les consulter.
Visual Basic facilite l'ajout des lignes nécessaires à la base de registres de Windows. Nous n'avons donc pas à écrire de code ni à comprendre le format des entrées de la base de registres. Lorsque nous créons le fichier exécutable, Visual Basic l'enregistre automatiquement sur l' ordinateur[13].

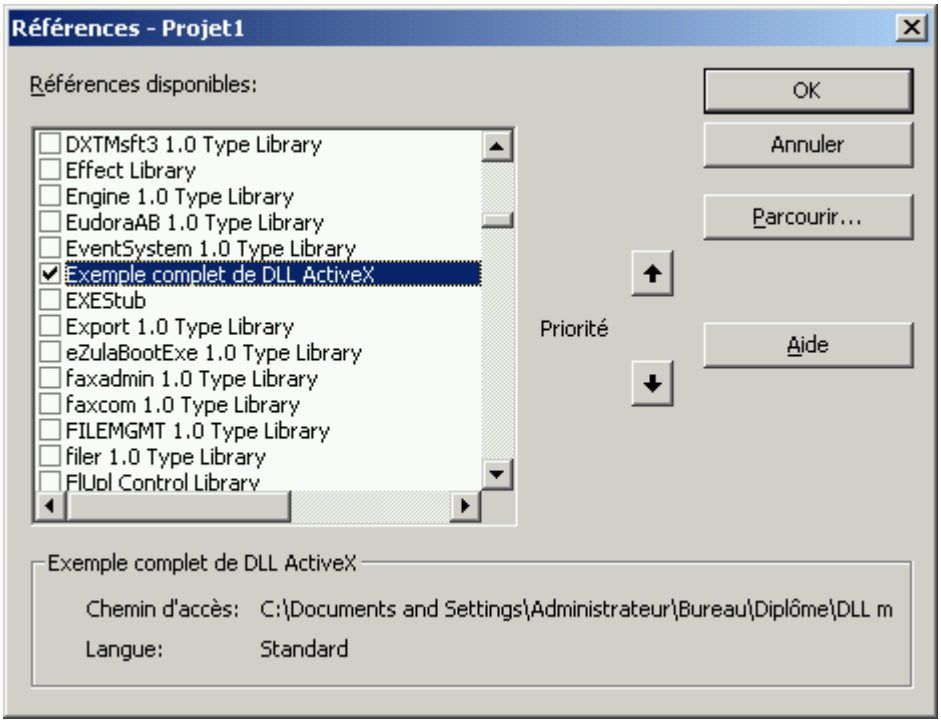
[13] Si l'application est distribuée dans le cadre d'une solution intégrée, l'inscription à la base de registres du système s'effectue pendant l'installation. Lors de l'utilisation de l'Assistant Empaquetage et déploiement de Visual Basic, l'inscription à la base de registres du système du composant s'intégrera automatiquement dans le processus d'installation.

Compilons donc notre DLL (Fichier → Créer ThingDemo.dll) en ayant auparavant eu soin de sélectionner ThingDemo dans la fenêtre d'explorateur de projets.

Testons maintenant notre composant binaire compilé. Supprimons donc `ThingDemo` du groupe de projet (Fichier → Supprimer le projet). Visual Basic affiche un message d'avertissement, car le projet `TestThing` contient une référence à `ThingDemo`. Cliquons sur Oui pour supprimer `ThingDemo` quoi qu'il en soit.
Lorsque nous supprimons `ThingDemo` d'un groupe de projets, Visual Basic recherche `ThingDemo.dll` dans la base de registres de Windows[14]. Si le fichier .dll existe, Visual Basic met automatiquement à jour la référence que nous avions définie au chapitre 4.

Pour tester le bon fonctionnement, il suffit donc d'exécuter le tout (Ctrl+F5).

Il est maintenant également possible d'utiliser `ThingDemo.dll` dans une autre copie de Visual Basic. Pour s'en convaincre, ouvrons une nouvelle instance de VB, créons un simple projet EXE standard, puis faisons une référence à notre DLL (Projet → Références) en cochant « Exemple complet de DLL ActiveX » (soit ce que nous avions saisi au point 3 dans les propriétés du projet). Nous pouvons alors ajouter un code pour créer les objets `Thing` et `Dialogs` et appeler leurs propriétés et leurs méthodes.



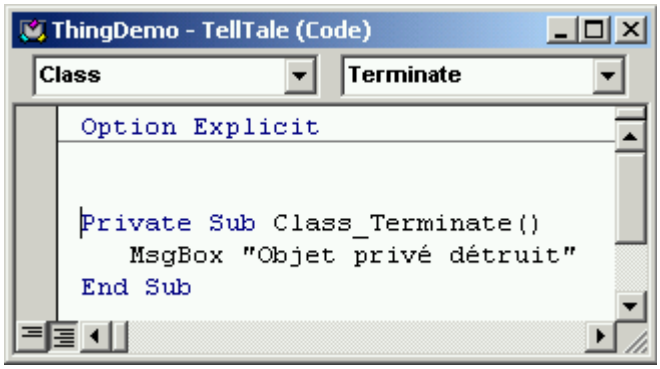
[14] Nous étudierons ce point précisément plus loin.

4.11 Encore plus de références circulaires et arrêt des composants

La procédure suivante décrit la manière dont Visual Basic décharge un composant in-process après que le client ait libéré toutes les références à ses objets. Elle explique également comment les références circulaires peuvent empêcher ce déchargement, et met en évidence une différence importante entre les objets publics et privés.

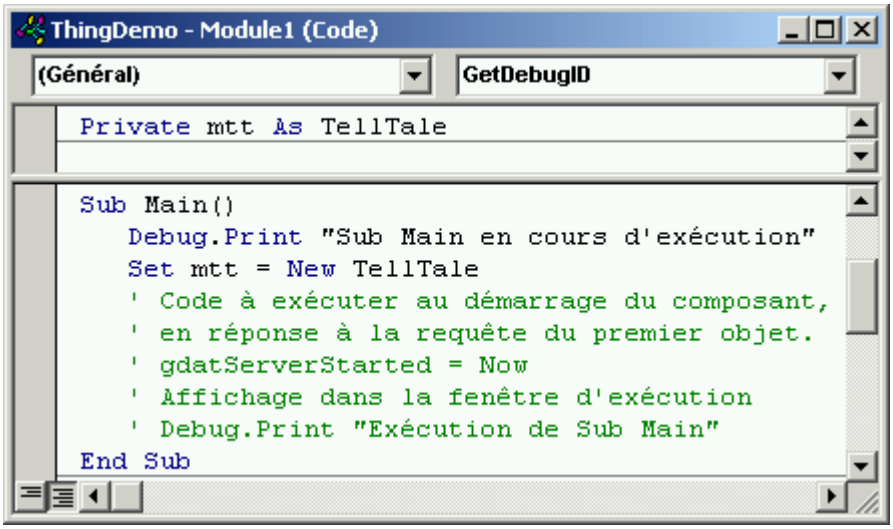
La procédure ne peut être exécutée qu'à l'aide du composant compilé et du projet test compilé, car Visual Basic n'effectue jamais de déchargement pendant qu'un projet de composant in-process est exécuté dans l'environnement de développement.

Observons comment se déroule le déchargement de la DLL. Pour ce faire, ajoutons un nouveau module de classe à `ThingDemo`, et appelons-le `TellTale`, et affectons la valeur `Private` à sa propriété `Instancing`. Ajoutons maintenant dans son événement `Terminate` le code suivant :



Un objet privé ne peut être créé par des clients et ne doit jamais leur être transmis. Comme nous le constaterons, les objets privés n'empêchent pas le déchargement des composants in-process et un client qui utilise une référence à un objet privé une fois que son composant est déchargé subira un échec de programme catastrophique.

Ajoutons le code suivant au `Module1` de `ThingDemo` :



Une fois les projets compilés, les instructions `Debug` ne peuvent être utilisées pour montrer ce qui se passe en interne. En revanche, l'objet `TellTale` nous donnera une indication de ce qui se passe dans le composant compilé.

Dans le menu Fichier, cliquons sur Créer le groupe de projets pour compiler à la fois `ThingDemo` et `ThingTest`. Depuis Windows, exécutons `ThingTest`.

Après avoir fermé la boîte de dialogue modale, cliquons sur Créer un objet Thing temporaire. Tapons « temp », puis validons.
Comme nous l'avons constaté précédemment, l'objet temporaire ne dure pas très longtemps. Dans le présent cas, il n'est créé que pour charger `ThingDemo.dll` et exécuter `Sub Main`. Dans le cadre de cette procédure, la question intéressante est de savoir ce qui se passe lorsque les objets `Things` sont partis.

Après quelques minutes (deux environ, bien que cette durée varie selon la fréquence du temps d'inactivité), une boîte de dialogue apparaît, affichant « Objet privé détruit ». Que s'est-il passé ? Il n'existait aucune référence aux objets publics, c'est pourquoi Visual Basic tentera de décharger le composant in-process.

Quand la DLL est déchargée, Visual Basic libère la mémoire qu'il utilisait, y compris les variables contenant des références aux objets privés. En conséquence l'objet `TellTale` est détruit et nous obtenons une indication visuelle du déchargement de la DLL.

Si, à ce moment, nous créons un nouvel objet `Thing`, nous observerons qu'il se produit une courte pause pendant que la DLL est rechargée.

Passons maintenant aux références circulaires. Cochons la case Référence à l'objet, puis cliquons sur Créer un objet Thing TEMPORAIRE pour créer un objet `Thing` avec une référence à lui-même.
Cette fois, inutile d'attendre des heures : la DLL ne se déchargera pas, car la référence à l'objet `Thing` empêche Visual Basic de la décharger.

Visual Basic ne peut faire la différence entre une référence interne à un objet public et une référence externe (client) à un objet public, de sorte qu'il ne lui reste qu'une seule alternative : conserver la DLL chargée.

En fermant la feuille principale de `ThingTest`, l'objet privé est à nouveau détruit, car le déchargement de l'application client décharge également tous les composants in-process qu'elle utilise.

En résumé, pour créer un composant efficace, nous devons renoncer à exercer un contrôle sur sa durée de vie : notre composant est démarré automatiquement lorsqu'un client demande un objet et il devrait normalement s'arrêter lorsque tous les clients libèrent les objets qu'ils utilisaient — et pas une seconde plus tôt.

La raison à cela est que les objets fournis par notre composant deviennent partie intégrante des applications client qui les utilisent. Un client qui utilise un de nos objets doit pouvoir disposer de cet objet jusqu'à ce qu'il n'en ait plus besoin.

Toujours selon ce principe, si les clients libèrent toutes les références aux objets fournis par notre composant, celui-ci devra s'arrêter et libérer la mémoire ainsi que toutes les ressources qu'il exploitait.
Par conséquent, pour créer un composant efficace, nous devons respecter certaines directives dont les deux plus importantes sont les suivantes :

- Ne jamais provoquer l'arrêt du composant lorsque des applications client ont toujours des références aux objets.
- Ne jamais conserver le composant en mémoire après que toutes les références aux objets ait été libérées.

5 - Cycle de vie d'une DLL ActiveX

5.1 Dans la salle des machines…

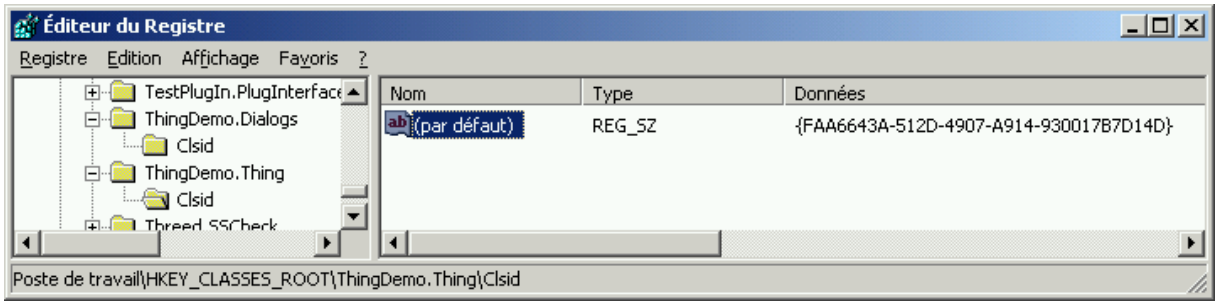
Nous allons maintenant rentrer dans la « salle des machines » de Windows, afin de mieux comprendre comment se déroule le cycle de vie de la DLL que nous venons de créer. Si Visual Basic s'occupe d'enregistrer la DLL lorsqu'on la compile – et donc nous évite d'avoir à le faire nous même par la commande « regsvr32.exe ThingDemo » - que se passe-t-il dans les décors ? C'est ce que nous allons – entres autres – expliquer..

Nous travaillerons avec l'éditeur de la base de registres[15] : prudence donc, puisqu'une manipulation malencontreuse pourrait sérieusement endommager le système, voire le rendre impossible à redémarrer !

L'éditeur de registres d'un système d'exploitation permet de conserver toutes les informations relatives à la configuration de l'OS et des applications. La plupart des paramètres de ces derniers peuvent être modifiées via certains logiciels, mais les développeurs peuvent les visualiser, voire les modifier directement dans le registre.

Note : Toutes les valeurs telles que les GUID, CLSID, ou nom de répertoires peuvent être différentes de celles que nous montrerons ici !

Ouvrons donc Regedit.exe[16] (via Exécuter du menu Démarrer), et cherchons dans HKEY_CLASSES_ROOT notre classe `ThingDemo`.



La base de registres est construite hiérarchiquement. A chaque niveau de cette hiérarchie est associé une clé (similaire à un répertoire sur disque). Chaque clé possède un nom (qui peut être donné par défaut, comme le montre la figure ci-dessus) et autant de valeurs que nécessaire. La clé `ThingDemo.Thing` n'a pour sa part pas de nom, mais par contre contient la sous-clé Clsid. Ce dernier terme est l'abréviation de class ID, ou identificateur de classe. La valeur de cette sous-clé tient en une chaîne représentant un nombre de 16 bits, nombre permettant d'identifier de manière unique l'objet. Lorsqu'une

[15] Windows 2000 fournit deux versions de l'Éditeur du Registre : Regedt32.exe (32 bits) et Regedit.exe (16 bits). Microsoft recommande d'utiliser Regedit.exe uniquement pour ses fonctionnalités de recherche, et de recourir à Regedt32.exe lorsqu'il est nécessaire de modifier le Registre.

[16] Dans un environnement partagé par plusieurs utilisateurs (tel Windows NT4 à l'EIVD), l'administrateur désactive (généralement) l'accès à la base de registres (pour des raisons évidentes de sécurité…). Afin d'y accéder tout de même, il suffit de créer un fichier texte et d'y placer les instructions :

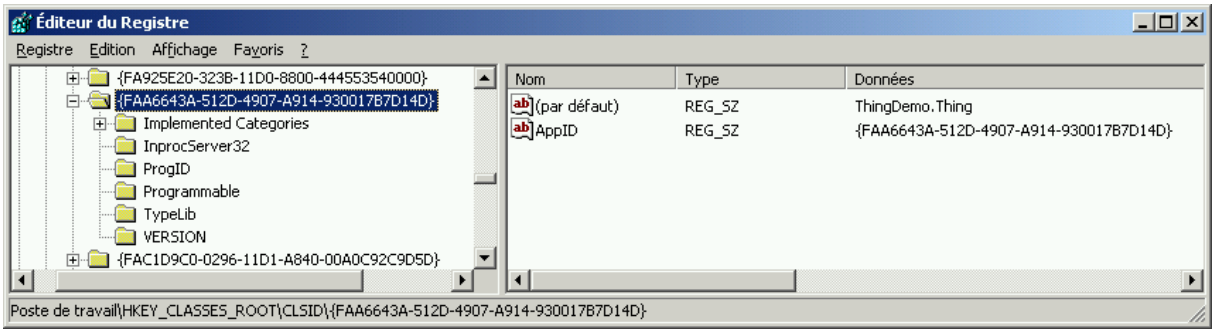
```
REGEDIT4
[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\Explorer\RestrictRun]
[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Policies\System]
"DisableRegistryTools"=dword:00000000
```

Sauvegarder le document avec l'extension .reg, puis le lancer.

application appelle cet objet par son nom – par exemple lors d'un appel à la fonction `CreateObject` – Windows sait où le retrouver grâce à ce numéro.

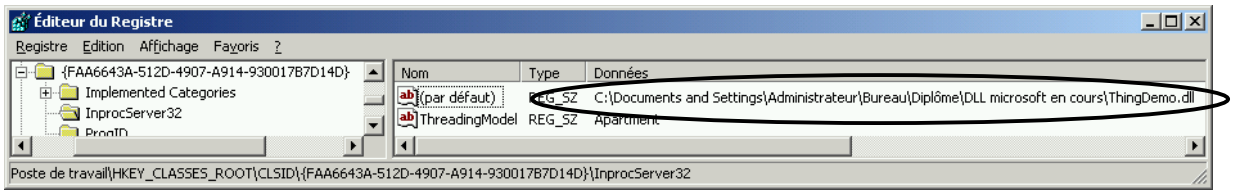
Une fois l'identificateur de cet objet trouvé, il nous faut pister le fichier exécutable (EXE ou DLL) qui sait comment créer l'objet et qui contient le code implémentant ses interfaces. Allons donc du côté de HKEY_CLASSES_ROOT\CLSID, et cherchons le Clsid de la valeur donnée par l'objet `ThingDemo.Thing`.

Voilà déjà plus d'informations ! La valeur {FAA66…} de la clé du Clsid est le nom de l'objet `ThingDemo.Thing`. Sous cette clé, on trouve plusieurs sous-clé :



- *Implemented Categories*
  Contient une ou plusieurs sous-clés qui sont autant de GUID pour les clés inscrites dans l'entrée HKEY_CLASSES_ROOT\Component Categories. Chaque valeur sous cette clé définit quelque chose sur l'objet. En tant que développeur VB, nous n'avons pas à nous en soucier.

- *InProcServer32*
  Cette entrée indique au système quelle DLL contient le code implémentant l'objet lors de l'exécution. C'est clairement l'information la plus importante, qui permet de visualiser comment une application peut localiser la DLL implémentant l'objet[17].



- *ProgID*
  L'identificateur de l'objet. Autrement dit, le nom utilisé par un programme pour identifier l'objet (ThingDemo.dll dans notre cas).

- *Programmable*
  Un champ qui ne semble pas servir à grand chose, puisqu'il indique que l'objet est programmable.

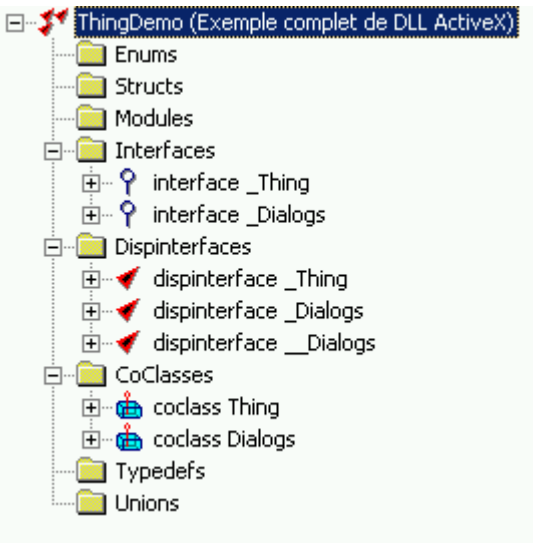
- *TypeLib*
  Cette clé contient la valeur du GUID de la librairie de type, librairie définissant les interfaces supportées par l'objet.

- *Version*

[17] Lorsqu'un EXE ActiveX est utilisé pour implémenter un objet, il est référencé par un champ nommé LocalServer32.

Le numéro de version de l'objet, sans rapport (du moins d'après ce qu'en sait l'auteur) avec le numéro de version de la DLL dans Visual Basic.

Il nous reste à creuser la librairie de types (TypeLib).
Pour visualiser cette librairie, l'éditeur de registre ne nous apporte rien de vraiment alléchant (les entrées de HKEY_CLASSES_ROOT\TypeLib ne font que référencer la DLL contenant les librairies de types). Par contre, Visual Basic fournit un outil nommé OLE View qui permet de découvrir les objets de manière très précise, et de sauvegarder les informations de la librairie de types dans le format IDL, dont nous avons parlé dans le chapitre 1. Ouvrons donc notre dll (File → View TypeLib…)



L'interface utilisée pour l'objet Thing se nomme _Thing. Notons la ligne qui la définit :

```
interface _Thing : IDispatch {
```

Le caractère ":" indique que cette interface est dérivée de l'interface `Idispatch`.

Notons également que la variable déclarée publique dans la classe Thing (`Public Name As String`) est implémentée comme deux fonctions – une avec l'attribut propget, la seconde avec l'attribut propput – correspondant aux fonctions `Property Get`, respectivement `Property Let`.

```
[id(0x40030000), propget] HRESULT Name([out, retval] BSTR* Name);
[id(0x40030000), propput] HRESULT Name([in] BSTR Name);
```

Le reste des informations disponibles dans l'application OLE View n'est compréhensible qu'aux véritables Gourous d'OLE…

Maintenant que nous avons fait un bref tour de la salle des machines, remontons au poste de pilotage, pour enfin comprendre le cycle de vie d'un objet implémenté dans une DLL.

5.2 Cycle de vie d'une DLL

Au moment de l'inscription (Registration Time)

Avant qu'une application puisse accéder à l'objet `ThingDemo.Thing`, les informations le concernant doivent figurer dans la base de registres. Nous avons vu que Visual Basic effectue cette opération automatiquement lors de la compilation. Lors de la distribution d'une application nécessitant une ou plusieurs DLL, les composants sont inscrits (registered) soit par le programme d'installation (si il y en a un), soit en utilisant le programme regsrv32.exe.
Les DLL de Visual Basic contiennent les fonctions exportées `DllRegisterServer` et `DllUnregisterServer`, qui sont appelée pour effectuer, respectivement effacer l'inscription de la DLL. Ce sont ces fonctions qui sont appelées par regsvr32.exe ou par le programme d'installation.

Au moment de la conception (Design Time)

Une référence à l'objet `ThingDemo.Thing` a été faite :

→ VB peut utiliser l'identificateur du programme (program ID) pour chercher dans le registre le CLSID de l'objet.

→ VB trouve dans le registre ThingDemo.dll.

→ VB lit la librairie de types de la DLL et connaît à présent les interfaces de l'objet.

→ Lorsqu'on sauve un projet, l'identificateur TypeLib est stocké dans le fichier du projet, ainsi quand VB charge ce projet, il peut lire la librairies de types en premier et en extraire le CLSID de l'objet.

Au moment de la compilation (Compilation Time)

Puisque l'information de l'interface est disponible au moment de la conception (Design Time), tous les appels aux variables définies dans l'objet `Thing` peuvent être faits avec une liaison précoce[18] (early binding). VB peut donc compiler les appels aux fonctions de l'interface au lieu de dépendre de l'interface IDDispatch de l'objet.

Au moment de l'exécution (Runtime)

Lorsqu'une application essaie de créer un objet Thing, elle utilise le CLSID de celui-ci en cherchant le nom de la DLL implémentant l'objet dans la base de registres. Ensuite :

→ VB charge la DLL.

→ VB utilise les fonctionnalités d'OLE pour créer l'objet comme désiré. Quand OLE crée cet objet, il retourne un pointeur sur l'interface `IUnknown` de l'objet, et VB utilise `QueryInterface` pour obtenir l'interface désirée (`_Thing` par exemple), qui est assignée à une variable de l'objet. L'objet est ensuite créé avec le compteur de référence à 1.

→ Les propriétés et les méthodes de l'interface de l'objet sont exécutées par appels directs des fonctions de l'interface (early binding).

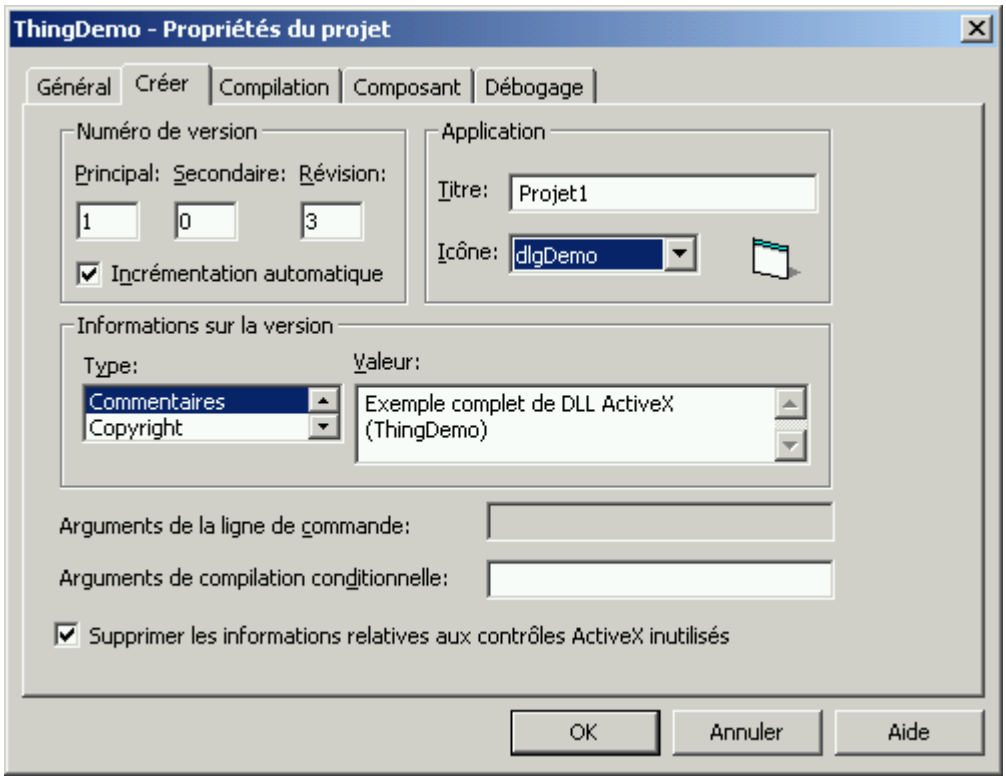
→ Lorsque le projet est fermé ou que la variable objet est mise à `Nothing`, le compteur de référence de l'objet passe à 0. La DLL Thing.dll le remarque et libère la mémoire de l'objet. Quand tous les objets implémentés par la DLL sont libérés, la DLL peut être déchargée.

[18] La liaison précoce apparaît lorsque VB est capable de déterminer le type de l'objet auquel on veut accéder au moment de la conception, et accélère le temps de traitement du programme puisque nécessite moins de travail en exécution pour déterminer le type de l'objet. Pour implémenter la liaison précoce, les variables doivent être déclarées d'un type spécifique (`Recordset` plutôt qu'`Object` par exemple).
Par opposition, la liaison tardive (late binding) apparaît lorsque VB est incapable de déterminer le type de l'objet (déclaré `Variant` par exemple).

5.3 Gestion des versions des DLL

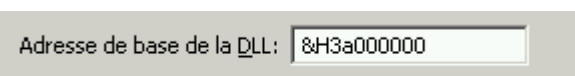
Un des problèmes rencontrés lors de l'utilisation de librairies de codes, telles les DLL ActiveX, est qu'il est difficile de parcourir le code source du fichier pour identifier ce qu'il contient… L'onglet Make de la boîte Propriétés du projet nous permet de définir un ensemble d'informations décrivant notre composant. Ces informations sont par la suite visibles pour l'utilisateur dans les propriétés du fichier de la DLL.

Une des informations la plus importante est le numéro de version, permettant un usage approprié de la bonne DLL lors d'un développement ultérieur. Ainsi, en cliquant sur le bouton « Incrémentation automatique », ce numéro de version sera automatiquement incrémenté après avoir créé une nouvelle copie du fichier DLL. Ceci nous permet de compiler et de tester – dans l'environnement de développement – nos objets autant de fois que nous le désirons, puis de « geler » le numéro de version lorsque la DLL est prête pour l'usage.



Le groupe Informations de version nous permet de spécifier des valeurs pour le nom de la compagnie, une description du fichier, les copyrights et le nom du produit. Il est également possible de saisir des commentaires. Toutes ces données sont ensuite visibles depuis les propriétés du fichier depuis Windows.

Certains auteurs recommandent de changer la valeur de l'adresse de base de la DLL, pour la raison suivante : chaque fichier DLL créé pour une application devrait avoir une adresse de base différente. Sinon, Windows devra reloger la DLL en cas de conflit avec une autre DLL active dans l'espace d'adressage. Il est donc conseillé de changer la valeur donnée par défaut (&H11000000) à une autre valeur (&H3A000000 par exemple), afin d'optimiser le temps pris pour charger la DLL en exécution. De même, on utilisera une adresse unique pour chaque DLL afin d'éviter ce type de conflit. Microsoft à ce sujet conseille même d'utiliser un bon générateur de nombres aléatoires…



Bibliographie critique et liens Internet

- Visual Basic developer's guide to COM and COM+, Wayne S. Freeze, Sybex 2000, 460 pages.
  L'ouvrage qui permet le mieux (parmi ceux cités dans cette bibliographie) de comprendre les technologies de composants de Microsoft, et met en relation simplement et efficacement les différents concepts. La partie consacrée à DCOM est malheureusement très courte, et n'offre qu'une petite introduction au protocole. L'auteur traite également COM+, MSMQ, et IMDB.

- Developping COM/ActiveX components with Visual Basic 6, Dan Appleman, Sams 1999, 860 pages.
  Cet ouvrage s'adresse avant tout aux développeurs professionnels sur VB, et montre les meilleures techniques pour développer des composants COM et DCOM. D'un niveau très nettement plus élevé que le titre précédent, cette « bible » (presque 900 pages !) passe en revue la conception d'objets COM mais aussi et surtout les mécanismes internes lors de l'exécution des composants. C'est à l'aide de cet ouvrage que j'ai réalisé une grande partie des explications techniques.

- http://msdn.microsoft.com, le lien INCONTOURNABLE pour développer sous plate-forme Windows.

- http://www.microsoft.com/com/tech/com.asp, la page de Microsoft pour tout ce qui concerne la technologie COM.