

Les interfaces d'objet sous Delphi

par **Laurent Dardenne** ([Contributions](#)) >

Date de publication : 28/01/2007

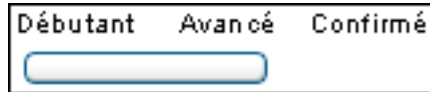
Dernière mise à jour : 03/02/2007

Vous trouverez dans ce tutoriel les bases nécessaires à la compréhension du fonctionnement et de la manipulation des interfaces d'objet sous Delphi, communément appelé Interface.

Je tiens à remercier **Sébastien Doeraene** pour sa relecture attentive et ses remarques pertinentes ainsi que Q1130 pour ses corrections orthographiques.


- 1 - Public concerné
- 2 - Les sources
- 3 - Rappels
 - 3-1 - Méthode abstraite
 - 3-2 - Classe abstraite
- 4 - Qu'est-ce qu'une interface ?
- 5 - A quoi servent-elles ?
- 6 - Comment ça marche ?
- 7 - L'interface IInterface
- 8 - L'interface inconnue (IUnknown)
 - 8-1 - Un compteur de références
 - 8-2 - La méthode QueryInterface
 - 8-3 - La méthode _AddRef
 - 8-4 - La méthode _Release
- 9 - Détails d'implémentation d'une interface
 - 9-1 - Les problématiques de transtypage des interfaces
- 10 - TInterfacedObject, l'implémentation de base de IUnknown
- 11 - Les problématiques du comptage de référence
 - 11-1 - Règle de portée
 - 11-2 - Rappel sur le transtypage
 - 11-3 - Mixage d'usage de référence et d'instance de classe
 - 11-4 - Signature de méthode
- 12 - Rappel des restrictions dans la déclaration d'une interface
 - 12-1 - Propriétés d'interface
- 13 - Appel polymorphique
 - 13-1 - Procédure générique
- 14 - Héritage multiple
 - 14-1 - L'héritage multiple d'interface
 - 14-2 - Multi-héritage de classe
- 15 - Collision de noms
 - 15-1 - Entre une classe et une interface
 - 15-2 - Entre deux interfaces distinctes
 - 15-3 - Entre des interfaces héritées
- 16 - Interroger l'arbre d'héritage d'une interface
- 17 - Les interfaces sous Delphi .NET
- 18 - Liens

1 - Public concerné



Testé sous Delphi 2006.

Version 1.0

 *La connaissance des principes de base de la programmation orientée objet sous Delphi est un pré requis à la compréhension de ce tutoriel.*

2 - Les sources

Les fichiers sources des différents exemples :

FTP.

HTTP.

L'article au format PDF.

3 - Rappels

3-1 - Méthode abstraite

Une **méthode abstraite** est une méthode virtuelle ou dynamique n'ayant pas d'implémentation dans la classe où elle est déclarée. Son implémentation est déléguée à une classe dérivée.

Un exemple de code

3-2 - Classe abstraite

Une classe abstraite est une classe n'ayant pas d'implémentation et ne pouvant être instanciée. Son implémentation est déléguée à une classe dérivée. Avant Delphi 2006 une classe abstraite contenait uniquement des méthodes abstraites, **depuis Delphi 2006** le mot clé **Abstract** peut être associé à une définition de classe

4 - Qu'est-ce qu'une interface ?

Ici le mot interface n'a aucun rapport ni avec une interface utilisateur (IHM) ni avec la section *interface* d'une unité Delphi.

Une interface est une abstraction. Elle est une spécification formelle de classe et est utilisée pour définir la limite entre sa spécification et son implémentation, on laisse ainsi aux classes la liberté de son implémentation.


En d'autre termes une interface est le **quoi** et une **classe** définit le **comment**.

Au niveau du langage Delphi, une interface est un type semblable à une classe abstraite, elle ne contient que des méthodes abstraites par défaut, il n'est pas nécessaire de spécifier le mot clé **abstract** dans leurs déclarations.

Une interface ne décrit aucune structure de donnée car elle ne peut contenir aucun attribut, elle n'a aucune donnée.

En revanche elle peut contenir des propriétés aux travers d'appel de méthodes d'accesseurs (Set et Get).

Elle est une possibilité **d'encapsulation** complète.

 *Les interfaces Delphi ressemblent fortement à celles de Java. Il y a cependant quelques différences au niveau de l'implémentation, en particulier la forte orientation COM des interfaces.*

5 - A quoi servent-elles ?

Une interface définit une aptitude à faire quelque chose, par exemple une itération sur un ensemble de données, ou un comportement de déplacement dans un espace 2D.

Un des objectifs des interfaces est de diminuer le couplage entre deux classes, c'est à dire le degré de dépendance d'une classe par rapport à une autre, ce qui facilite la maintenance.

L'interface permet de ne présenter au client que ce qui l'intéresse, dans ce tutoriel nous l'illustrerons en abordant les méthodes pouvant être liées au déplacement d'un être ou d'une chose.

L'usage d'une interface à la place d'une classe, permet de modifier l'implémentation de l'abstraction, ici le déplacement, sans impacter la ou les classes utilisatrices, l'interface faisant office d'écran.

Prenons un exemple, généralement la notion de déplacement, commune aux êtres ou aux objets, tombe sous le sens en revanche sa spécialisation au sein de la classe *serpent* ou *flèche d'arc* est radicalement différente. Une interface permettrait ici, dans un contexte précis, de s'intéresser à l'essentiel : **ce que ça fait et pas comment ça le fait**.

En quelque sorte une interface répond à la question que l'on pourrait poser à une classe : *Est-ce que tu sais faire ça ?*

La réponse qui nous importe ici est **oui** ou **non**, et c'est tout. La manière de le faire, c'est à dire comment est rempli le contrat, nous importe relativement peu.

Le terme de contrat est souvent employé pour définir la convention par laquelle une interface interagit avec une classe mais ce qui importe c'est plus le respect de ce que l'interface est sensée faire que l'acte qui enregistre cette convention. Dans la vie on peut prendre des engagements ou signer tous les contrats que l'on veut, rien ne nous empêche de ne pas les respecter. Je serais tenté de dire plus simplement qu'une interface respecte toujours l'engagement pris sur le comportement qu'elle propose et dans le cas contraire la sanction est simple et sans appel, la compilation échoue.

6 - Comment ça marche ?

Chargez le projet interface1.

Par convention, le nom d'une interfaces est préfixé par la lettre **I** en majuscule et son nom définit clairement son comportement.

Déclarons une interface proposant la capacité de se déplacer :

```
Type
IDeplacable =Interface
  Procedure Deplace; // Méthodes abstraites par défaut
  Procedure Arrêt;
end ;
```

On utilisera, par exemple la méthode *Deplace* non pas via une instance de classe mais au travers d'une variable du type de l'interface souhaité, on parlera donc dans ce cas de référence d'interface.

Sachez qu'une variable d'interface, ici la variable *MonInterface*, est initialisée à **Nil** lors de sa déclaration :


```
Procedure TestInterface(UneInterface : IDeplacable);
begin
  If assigned(UneInterface)
  then Writeln('L'interface est assignee.')
  else Writeln('L'interface n'est pas assignee.');
```

```
var MonInterface : IDeplacable;
begin
  TestInterface(MonInterface);
end.
```

Créons une interface en utilisant le code suivant :

```
var MonInterface : IDeplacable;
begin
  MonInterface:=IDeplacable.Create;
end.
```

malheureusement ce code provoque l'erreur de compilation : *"Type record, object ou class requis"*.

 *La réalisation d'une référence d'interface ne peut se faire en dehors de celle d'une instance de classe. Pour agir sur une interface on doit impérativement créer une instance d'une classe implémentant le type d'interface attendue.*

Les classes sont donc responsables de l'implémentation de la ou des interfaces qu'elles supportent.

Afin d'utiliser notre interface **IDeplacable** nous devons déclarer une classe l'implémentant :

```
TChose= Class(TObject, IDeplacable)
  Distance : Integer
end;
```

Le premier type déclaré doit être impérativement une classe, ici **TObject**, suivie d'une ou plusieurs interfaces, ici **IDeplacable**.

Ajoutons la création de l'instance de la classe **TChose** :

```
begin
  TestInterface(MonInterface);
  UneChose:=TChose.Create;
  readln;
end;
```

La compilation du code précédent nous renvoie, entre autres, les erreurs suivantes :

```
Identificateur non déclaré : Deplace
Identificateur non déclaré : Arret
```

Elles confirment que les méthodes de l'interface **IDeplacable** sont bien abstraites et doivent être implémentées par la classe **TChose**. Les simples déclarations des méthodes de l'interface sont insuffisantes, dans ce cas on retrouve une erreur classique : *"Déclaration forward ou external non satisfaite : ' TChose.Deplace"*.


Implémentons donc les deux méthodes de l'interface :

```
TChose= Class(TObject, IDeplacable)
  Distance : Integer;
  //IDeplacable
  Procedure Deplace(Const Param:String);
  Procedure Arret;
  //TChose
End;

{ TChose }
procedure TChose.Arret;
begin
  Writeln('Arret : IDeplacable de TChose ');
end;

procedure TChose.Deplace(const Param: String);
begin
  Writeln('Deplace : IDeplacable de TChose ');
end;
```

Ici la compilation nous renvoie la même erreur mais pour les méthodes nommées **QueryInterface**, **_AddRef** et **_Release**. Bien que nous respections le contrat de l'interface **IDeplacable**, notre classe est soumise au respect du contrat de l'interface héritée **Interface**. Du fait que, comme une classe, une interface hérite de toutes les méthodes de ces ancêtres on doit implémenter **toutes** ces méthodes.

 *Il reste possible de ne pas implémenter totalement une méthode d'interface, c'est à dire que le corps de la méthode peut ne rien faire.*

7 - L'interface IInterface

IInterface est l'ancêtre de base de toutes les interfaces, les déclarations suivantes :

```
type
  IDeplacable = interface

  IDeplacable = interface(IInterface)
```

sont donc identiques.

Voici la déclaration de l'interface **IInterface** :

```
type
  IInterface = interface
    ['{00000000-0000-0000-C000-000000000046}']
    function QueryInterface(const IID: TGUID; out Obj): HRESULT; stdcall;
    function _AddRef: Integer; stdcall;
    function _Release: Integer; stdcall;
  end;
```


Elle permet de gérer ou créer des interfaces COM sous Delphi. On peut également voir dans le code source de l'unité *System.pas* la déclaration suivante

```
IUnknown = IInterface;
```

Les trois méthodes de **IInterface** sont donc celles de **IUnknown**. Sous Delphi hériter de **IUnknown** au lieu de **IInterface** informe le compilateur que l'interface doit être compatible avec les objets COM.

8 - L'interface inconnue (IUnknown)

Chargez le projet Interface2.

L'interface  **IUnknown** est l'ancêtre des interfaces sous COM et possède son propre identifiant d'interface (**IID**), chaque objet COM doit l'implémenter. Les méthodes de l'interface **IUnknown** permettent de gérer le cycle de vie des objets et d'obtenir les autres interfaces gérées par le composant.


Dans la déclaration de **IInterface**, la partie optionnelle `['{00000000-0000-0000-C000-000000000046}']` spécifie un **FAQ GUID** qui permet d'identifier de manière unique une interface. Ce GUID est nécessaire pour l'interrogation d'interface via l'appel à la méthode **QueryInterface**.

8-1 - Un compteur de références

Les interfaces COM utilisent un compteur de références pour chaque instance d'objet afin de contrôler sa durée de vie. Une des raisons est que plusieurs clients peuvent utiliser une même référence d'interface.

Lorsqu'une référence d'interface est demandée, par exemple lors d'une assignation, le compilateur insère un appel à la méthode **_AddRef** qui incrémente de 1 son compteur de référence. Une fois que la référence d'interface n'est plus utilisée, par exemple lors de l'affectation à NIL, le compilateur insère un appel à la méthode **_Release** qui décrémente le compteur de référence, et si celui-ci est nul, le destructeur de l'instance d'objet implémentant l'interface est appelé.

On peut donc avoir plusieurs références d'une même interface pour une seule instance d'objet. Sans ce mécanisme il serait impossible de savoir quand détruire l'objet implémentant la ou les interfaces manipulées.

 *L'usage du compteur de référence comporte quelques pièges comme nous le verrons par la suite.*

8-2 - La méthode QueryInterface

Elle permet d'obtenir un pointeur d'interface pour l'interface identifiée par le paramètre **IID**. Si l'objet supporte l'interface requise, il est renvoyé dans le paramètre *Obj* et la méthode **QueryInterface** renvoie **S_OK**. Si l'objet ne supporte pas l'interface, **QueryInterface** renvoie **E_NOINTERFACE**.


8-3 - La méthode _AddRef

Incrémente le compteur de références de cette interface.

Une interface peut proposer ses services à plusieurs clients, chaque client devra donc posséder une référence sur cette interface.

8-4 - La méthode _Release

Décrémente le compteur de références de cette interface. Chaque client doit donc libérer la référence de chaque interface utilisée. Une fois le compteur de référence à zéro l'interface libère l'instance de classe sous-jacente.

 *Sous Delphi .NET ces méthodes sont facultatives sauf si votre interface doit être accessible via COM.*

9 - Détails d'implémentation d'une interface

Chargez le projet Interface3.

Revenons à notre code d'origine et utilisons la procédure *TestInterface* :

```
Procédure TestInterface(UneInterface : IDeplacable);
begin
  If assigned(UneInterface)
  then
  begin
    Writeln('L'interface est assignee. ');
    UneInterface.Deplace('Méthode d'interface');
  end
  else Writeln('L'interface n'est pas assignee. ');
end;
```

Appelons la avec en paramètre une instance de classe au lieu d'une référence d'interface.

```
UneChose:=TChose.Create;
TestInterface(UneChose);
readln;
```

On peut voir qu'il n'y a aucun problème particulier ni lors de la compilation ni lors de l'exécution. Par contre l'ajout, dans la procédure **TestInterface**, des lignes suivantes :

```
UneInterface.ProcedureDeTChose;
Inc(UneInterface.Distance,10);
```

provoque, lors de la compilation, l'erreur : "*Identificateur non déclaré*".

L'interface ne connaît rien de l'objet associé car comme il a été dit précédemment on s'attache à l'essentiel en respectant le contrat passé. En revanche l'implémentation d'une méthode d'une interface peut manipuler des données de l'instance de classe :

```
procédure TChose.Deplace;
begin
  Writeln('Deplace : IDeplacable de TChose');
  Inc(Distance,10);
end;
```


On peut donc dire que **TChose.Deplace** est égale à **IDeplace.Deplace** mais pas que **TChose** est égale à **IDeplace**.

On ne peut pas transtyper **IDeplace** en **TChose** en revanche **TChose** peut être transtypé en **IDeplace**, plus précisément il s'agit d'une conversion de type implicite. C'est ce qui se passe lors de l'appel à la procédure **TestInterface**, bien que cela soit le compilateur qui effectue cette opération à notre place. Il ne s'agit pas vraiment d'un transtypage car le type de l'interface est connu lors de la compilation. Il n'y a donc pas d'interrogation de la table des méthodes des interfaces (**IMT**).

Le code suivant crée un objet **TChose** puis récupère son interface **IDeplacable** qui permet l'appel de la méthode *Deplace* via la variable *MonInterface* et enfin décrémente le compteur de référence de l'interface l'interface.

```

UneChose:=TChose.Create;
MonInterface:=UneChose; //Affectation
                        //transtypage implicite lors de la compilation
MonInterface.Deplace;
MonInterface:=Nil;
UneChose.Free;
    
```

 Ici l'assignation de NIL dans la variable d'interface ne provoque aucune suppression de l'instance de classe car, bien qu'il soit en théorie à zéro, le compteur de référence n'est pas géré. Notez toutefois que l'appel des méthodes **_AddRef** et **_Release** est effectif.

```

Appel de TChose._AddRef
Deplace : IDeplacable de TChose
Appel de TChose._Release
    
```

9-1 - Les problématiques de transtypage des interfaces

Chargez le projet40.

Essayons de manipuler une interface **IDeplacable** sur une instance de la classe **TObject** :

```

var MonInterface : IDeplacable;
    UnObjet :TObject;

begin
    UnObjet:=TObject.Create;

{1} MonInterface:=UnObjet;
{2} if UnObjet is IDeplacable then writeln('Un objet est une interface IDeplacable. ');
{3} MonInterface:=UnObjet as IDeplacable;
{4} MonInterface:=IInterface(UnObjet) as IDeplacable;
{5} MonInterface:=IUnknown(UnObjet) as IDeplacable;
{6} if Not Supports(UnObjet,IDeplacable)
    then Writeln('la classe TObject ne supporte pas l'interface IDeplacable. ');

    UnObjet.Free;
    readln;
end.
    
```

La ligne 1 provoque l'erreur de compilation : *"Type incompatible IDeplacable et TObject."*

TObject ne supporte pas l'interface **IDeplacable**, le transtypage implicite ne peut se faire.

La ligne 2 provoque l'erreur de compilation : *"Opérateur non applicable à ce type d'opérande."*

L'opérateur **IS** est dédié aux classes uniquement.

La ligne 3 provoque l'erreur de compilation : *"Opérateur non applicable à ce type d'opérande."*

L'interface **IDeplacable** ne possède pas de GUID, le transtypage ne peut donc se faire.

La ligne 4 et 5 provoque l'erreur de compilation : *"Type incompatible IInterface et TObjet."*

TObjet ne supporte pas l'interface **IDeplacable**, le transtypage explicite à l'aide de l'opérateur **AS** ne peut se faire.

Ces erreurs de compilation mettent en évidence qu'une classe ne peut proposer que les interfaces qu'elle implémente, **TObjet** n'en propose aucune.

Il est possible de savoir si une classe supporte ou pas une interface donnée, Delphi propose à cette fin la fonction **Supports** :


```
if Not Supports(UnObjet, IDeplacable)
then Writeln('la classe TObjet ne supporte pas l''interface IDeplacable.');
```

Ce code provoque l'erreur de compilation : *L'interface 'IDeplacable' n'a pas d'identification d'interface.*

Ajoutons un GUID (un identifiant unique d'interface), par la combinaison de touche Shift-Ctrl-G :

```
IDeplacable=interface(IInterface)
[ '{62CAE27F-94C1-4A3D-B94F-F08FF36207D5}' ] // GUID nécessaire pour l'opération de cast
```

A partir de là l'appel de la fonction **Supports** réussit. En revanche les lignes de code 1 à 5 restent erronées (nous reviendrons sur l'erreur de la ligne 3).

 *Sous Delphi .NET l'opérateur **is** est valide sur une interface et l'usage de l'opérateur **as** ne nécessite pas de GUID, sauf si votre interface doit être accessible via COM.*

Chargez le projet Interface41.

Reprenons notre classe **TChose** pour opérer ce transtypage d'interface et ajoutons une seconde déclaration d'interface :

```
type
  IMesurable=interface(IInterface)
  Function Dimension:Integer;
  End;

...
var MonInterface : IDeplacable;
    Taille : IMesurable;
    UneChose: TChose;
begin
  UneChose:=TChose.Create;
  if Not Supports(UneChose, IDeplacable)
  then Writeln('la classe TChose ne supporte pas l''interface IDeplacable.');
```

```
MonInterface:=UneChose as IDeplacable;
```

Bien que l'interface **IDeplacable** possède un GUID, la dernière ligne provoque l'erreur de compilation : "*Opérateur non applicable à ce type d'opérande.*"

Ici on appelle une des fonctions **Supports** surchargée avec une instance de classe, cette méthode appelle en interne la méthode héritée **TObject.GetInterface** le traitement peut donc se faire sans problème puisqu'elle parcourt la liste des interfaces de l'instance *UneChose*.

Par contre l'opérateur **AS** implique que la classe concernée implémente la méthode **QueryInterface** afin d'obtenir un pointeur d'interface pour l'interface identifiée par le paramètre **IID** (c'est à dire *IDeplacable*).

Ce qui est bien le cas dans notre exemple! Mais que se passe-t-il donc ?

Essayons de modifier la déclaration de la classe **TChose** ainsi :

```
//TChose= Class(TObject, IDeplacable)  
TChose= Class(TInterfacedObject, IDeplacable)
```

La compilation réussie.

La raison en est que la classe **TInterfacedObject** implémente l'interface **IUnknown**, avec cette déclaration le compilateur sait que la classe **TChose** implémente bien la méthode *QueryInterface*.


Dans notre déclaration d'origine **IDeplacable** hérite de **IInterface** mais masque au compilateur cette information.

Déclarons explicitement cette interface dans la liste d'interfaces de notre classe:

```
TChose= Class(TObject, IInterface, IDeplacable)
```

Ici aussi la compilation réussie.

On peut donc ainsi créer une interface identifiée, supportant le transtypage et sans gestion du compteur de référence ce qui n'est pas le cas si on utilise la classe de base **TInterfacedObject** au lieu de **TObject**.

 **Attention l'usage de l'opérateur *As* sur une interface incrémente le compteur de référence :**

```
Appel de TChose.QueryInterface  
Appel de TChose._AddRef
```

Pour terminer abordons la gestion de la seconde interface qui dispose d'un GUID :

```
Taille:=UneChose as IMesurable;
```

Ce transtypage provoque à l'exécution l'exception **EIntfCastError**. On doit donc utiliser un bloc **Try..Except** pour la gérer

```
try
  Taille:=UneChose as IMesurable; // cf. la procédure interne à Delphi System._IntfCast
except
  On E :EIntfCastError do
    begin
      Writeln('Exception :');
      Writeln(E.Message);
    end;
end;
```


 Vous remarquerez qu'ici le compteur de référence n'est pas appelé.

```
Appel de TChose.QueryInterface
Exception :
Interface non supportée
```

10 - TInterfacedObject, l'implémentation de base de IUnknown

La classe **TInterfacedObject** implémente l'interface **IUnknown** et peut être utilisée comme base simple pour des classes compatibles COM car cette classe implémente un compteur de référence.

Dans Delphi, un descendant de **TInterfacedObject** n'a pas de [FAQ fabricant de classes](#) ni de CLSID(*CLaSS IDentifier*). Cela signifie qu'il ne peut être instancié qu'en appelant un constructeur.

 *Je n'aborderais pas cet aspect des interfaces car on entre dans le **modèle COM** qui est par ailleurs très bien expliqué dans des tutoriaux disponibles sur [Developpez.com](#).*

Nous utiliserons désormais les interfaces sans variable d'instance de classe ce qui implique une création différente :

```
var MonInterface : IDeplacable;  
    UneChose : TChose;  
  
begin  
    MonInterface:=TChose.Create; //Crée une interface  
    UneChose:=MonInterface;  
end;
```

Lors de la création de l'interface le compilateur ajoute un appel à **_AddRef**.

La seconde ligne provoque à la compilation l'erreur : *Type incompatible TChose et IDeplacable*. Ce qui confirme une fois de plus qu'une interface n'est pas une classe.

11 - Les problématiques du comptage de référence

Désormais l'implémentation de **IUnknown**, dans la classe **TInterfacedObject**, gère à notre place la destruction des interfaces. Bien qu'étant déchargé de cette opération il reste quelques erreurs à ne pas commettre. N'ayant plus la main sur l'implémentation des méthodes de **IUnknown**, utilisons le destructeur de la classe implémentant notre interface pour afficher le moment où elle est détruite.

11-1 - Règle de portée

Chargez le projet Interface42.

La portée d'une interface peut être globale au programme :

```
Procedure PorteeInterface;
begin
  Writeln('Appel de la procedure PorteeInterface. ');
  MonInterface.Deplace;
end;

begin
  MonInterface:=TChose.Create; //Créé une interface
  PorteeInterface;
  Writeln;
  Readln;
```

L'interface est ici supprimée en fin d'exécution du programme. Voyons ce qui se passe si on affecte la valeur **NIL** à l'interface :

```
MonInterface:=TChose.Create; //Créé une interface
MonInterface:=Nil;
PorteeInterface;
```

La suppression explicite de l'interface appelle le destructeur de l'instance l'implémentant, ce qui provoque une AV dans la procédure *PorteeInterface*.

La portée d'une interface peut aussi être locale à une méthode :

```
Procedure PorteeInterface2;
Var lInterface : IDeplacable;
begin
  Writeln('Appel de la procedure PorteeInterface2. ');
  lInterface:=TChose.Create; //Créé une interface
  lInterface.Deplace;
end;
```


Dans ce cas et une fois la procédure terminée, l'interface n'étant plus utilisée, le compilateur ajoute un appel à **IUnknown._Release** qui décrémente le compteur à zéro et déclenche l'appel **TChose.Destroy**.

L'appel suivant compile et s'exécute mais comporte un effet de bord :

```

Procedure PorteeInterface3;
begin
    Writeln('Appel de la procedure PorteeInterface3. ');
    IDeplacable(TChose.Create).Deplace;
end;
    
```

On s'aperçoit qu'il n'y a pas d'appel à **Destroy** ce qui provoque une fuite mémoire.

 *La création d'une interface doit se faire à l'aide d'une variable d'interface.*

11-2 - Rappel sur le transtypage

Comme nous l'avons vu précédemment l'opérateur **As** incrémente le compteur de référence, dans l'exemple suivant :

```


procedure ImpactDuTranstypageSurLeCompteurDeReference;
var UneReference : IUnknown;
    Interfacel : IDeplacable;

begin
    Writeln('Appel de la procedure ImpactDuTranstypageSurLeCompteurDeReference. ');
    Interfacel:=TChose.Create;
    UneReference:= Interfacel as IUnknown;
    Interfacel:=Nil;
    Writeln('Fin de la procedure. ');
end;
    
```

la destruction de l'interface intervient après l'affichage du message "*Fin de la procedure*" mais si on souhaite la supprimer avant cet affichage on doit forcer la libération des autres références :

```

Interfacel:=Nil;
UneReference := Nil; //Force la libération de l'interface
Writeln('Fin de la procedure. ');
    
```

 *Il est tout à fait possible de typer la variable UneReference en **Variant**, qui supporte les interfaces, mais dans notre contexte cela n'est pas nécessaire, de plus les traitements liés au type **Variant** augmentent le temps d'exécution.*

Le tutoriel intitulé  **Delphi reference counted interfaces** vous donnera le détail des appels effectués par le compilateur Delphi.

Voir aussi  **The Rules of the Component Object Model**.

11-3 - Mixage d'usage de référence et d'instance de classe

Chargez le projet interface43.

Dès lors que l'on utilise, sur un objet dérivé de **TInterfacedObjet**, une instance de classe **et** une référence d'interface, on se retrouve avec des problèmes insolubles.

Prenons le code suivant :

```
var MonInterface : IDeplacable;  
    UneChose : TChose;  
  
begin  
    UneChose:=TChose.Create;  
    MonInterface:=UneChose;  
  
    if Assigned(UneChose)  
    then FreeAndNil(UneChose);  
  
    readln;  
end.
```

Son exécution provoque, sur l'appel de **FreeAndNil**, une exception **EInvalidPointer** (Opération de pointeur incorrecte).

L'instance *UneChose* existe bien et son compteur de référence d'interface est à 1. On peut être tenté de supprimer la référence d'interface :

```
MonInterface:=UneChose;  
MonInterface:=Nil;
```

mais dans ce cas le compteur passe à 0 et la méthode **_Release** détruit l'instance. La suppression de l'appel de **FreeAndNil** règle ce problème me direz-vous, certes mais si ces différents appels sont répartis dans plusieurs méthodes c'est une autre histoire.

L'exception est provoquée par le code suivant :

```
procedure TInterfacedObject.BeforeDestruction;  
begin  
    if RefCount <> 0 then  
        Error(reInvalidPtr);  
end;
```

Un garde fou bien utile dans notre contexte.

Borland recommande de ne pas mixer l'usage d'objet et d'interface à cause de possibles effets de bord.

11-4 - Signature de méthode

Texte issu de la documentation de Delphi 7

*Le compilateur Delphi vous fournit l'essentiel de la gestion mémoire **l'interface** grâce à son implémentation de l'interrogation et du comptage de références de l'interface. Par conséquent, si vous utilisez un objet qui vit et meurt via ses interfaces, vous pouvez aisément vous servir du comptage de références en dérivant de **TInterfacedObject**. Si vous choisissez d'utiliser le comptage de références, vous devez faire attention à ne manipuler l'objet que sous la forme d'une référence d'interface et à être cohérent dans votre comptage de références.*

Par exemple :

```
procedure beep(x: ITest);
function test_func()
var
  y: ITest;
begin
  y := TTest.Create; // comme y est de type ITest, le compteur de références vaut 1
  beep(y); // l'appel de la fonction beep incrémente le compteur de références
             // et le décrémenté à son retour
  y.something; // l'objet est toujours là avec un compteur de références valant 1
end;
```

C'est la manière la plus claire et la plus prudente de gérer la mémoire et, si vous utilisez **TInterfacedObject**, elle est mise en #uvre automatiquement. Si vous ne respectez pas ces règles, votre objet peut disparaître inopinément, comme illustré dans le code suivant :

```
function test_func()
var
  x: TTest;
begin
  x := TTest.Create; // pas encore de compteur de références pour l'objet
  beep(x as ITest); // le compteur est incrémenté par l'appel de beep
                   // et décrémenté à son retour
  x.something; // surprise ! l'objet n'est plus là
end;
```

Remarque

Dans les exemples précédents, la procédure **beep**, telle qu'elle est déclarée, incrémente le compteur de références (appel de `_AddRef`) pour le paramètre. Par contre, les déclarations suivantes ne le font pas :

```
procedure beep(const x: ITest);
```

ou

```
procedure beep(var x: ITest);
```

Ces déclarations génèrent un code plus concis et plus rapide.

Vous ne pouvez pas utiliser le comptage de références dans un cas : si votre objet est un composant ou un contrôle contenu dans un autre composant. Dans un tel cas, le comptage de références ne peut pas être appliqué de manière cohérente : vous pouvez toujours utiliser les interfaces, mais sans utiliser le comptage de références car la durée de vie de l'objet n'est pas régie par ses interfaces.

En fait **TComponent** implémente **IInterface** en ne faisant rien dans `_AddRef` et `_Release` (si ce n'est renvoyer -1).

12 - Rappel des restrictions dans la déclaration d'une interface

Extrait du "Guide du langage Pascal Objet", chapitre 10-1. Copyright © 1983, 2001 Borland Software Corporation.

- *La liste des membres ne peut contenir que des méthodes et des propriétés ; les champs ne sont pas autorisés dans les interfaces.*
- *Une interface n'ayant pas de champ, les spécificateurs de propriété **read** et **write** doivent être des méthodes (puisque les champs ne sont pas utilisables).*
- *Tous les membres d'une interface sont publics. Les spécificateurs de visibilité et de stockage ne sont pas autorisés (il est par contre possible d'utiliser **default** avec une propriété tableau).*
- *Les interfaces n'ont ni constructeurs, ni destructeurs. Elles ne peuvent être instanciées, si ce n'est via des classes qui implémentent leurs méthodes. Il n'est pas possible de déclarer des méthodes comme étant **virtual**, **dynamic**, **abstract** ou **override**. Comme les interfaces n'implémentent pas leurs propres méthodes, ces directives sont dépourvues de sens.*

12-1 - Propriétés d'interface

Les propriétés déclarées dans une interface ne sont accessibles que dans des expressions de type interface : il n'est pas possible d'y accéder via des variables de type classe.

De plus, les propriétés d'interface ne sont visibles que dans les programmes où l'interface est compilée. Par exemple, sous Windows, les objets COM n'ont pas de propriété.

13 - Appel polymorphe

Chargez le projet Polymorphe

Les interfaces facilitent les traitements polymorphiques sur des classes ne possédant pas d'ancêtres communs.

Extrait de l'aide en ligne de Delphi :

L'utilisation d'interfaces vous permet de séparer la manière d'utiliser une classe de la manière dont elle est implémentée. Deux classes peuvent implémenter la même interface sans descendre de la même classe de base. En obtenant une interface à partir d'une classe, vous pouvez appeler les mêmes méthodes sans avoir à connaître le type de la classe. Cette utilisation polymorphe de la même méthode pour des objets sans rapport entre eux est possible car les objets implémentent la même interface.

Voir l'exemple du cours de [Didier Mailliet](#) en page 144-116.

```
type
  IDeplacable=interface(IInterface)
  ['{62CAE27F-94C1-4A3D-B94F-F08FF36207D5}']
  Procedure Deplace;
End;

TChose= Class(TInterfacedObject, IDeplacable)
  Procedure Deplace;
end;

TEtre= Class(TInterfacedObject, IDeplacable)
  Procedure Deplace;
end;
...
var InterfaceDeplace :IDeplacable;

begin
  InterfaceDeplace:=TChose.Create;
  InterfaceDeplace.Deplace;
  InterfaceDeplace:=Nil;

  InterfaceDeplace:=TEtre.Create;
  InterfaceDeplace.Deplace;
  InterfaceDeplace:=Nil;
end.
```

13-1 - Procédure générique

A partir du moment où, au travers d'interface, on manipule des classes ne possédant pas d'ancêtre communs, il devient possible d'avoir des traitements communs :

```
procedure DeplaceTout(AInterface: array of IDeplacable);
var
  I: Integer;
begin
  for I := Low(AInterface) to High(AInterface) do
    AInterface[I].Deplace;
end;
```

```
var InterfaceChose,  
    InterfaceEtre : IDeplacable;  
  
begin  
    InterfaceChose:=TChose.Create;  
    InterfaceEtre:=TEtre.Create;  
    DeplaceTout ([ InterfaceChose, InterfaceEtre ] );  
    InterfaceChose:=Nil;  
    InterfaceEtre:=Nil;  
  
    readln;  
end.
```

14 - Héritage multiple


14-1 - L'héritage multiple d'interface

Le code suivant n'est pas possible sous Delphi Win32 :

```
IMoteur=Interface
  procedure Demarrer;
  Procedure Stopper;
end;

ICarosserie=Interface
  Procedure GetPortes;
end;

IVoiture=Interface(IMoteur,ICarosserie)
  Procedure GetPlace;
end;
```

 *Ce code d'héritage multiple d'interfaces, aussi nommé combinaison d'interfaces par certains, est en revanche valide sous Delphi .NET.*

14-2 - Multi-héritage de classe

Le langage Delphi n'autorise pas l'héritage multiple, une classe ne peut hériter que d'une seule classe, mais en revanche, le langage Delphi autorise l'héritage de comportements multiples au travers des interfaces.

On peut aussi parler de typage multiple c'est à dire qu'un objet de la classe **TChose** est en même temps de type **TChose**, **Iinterface**, **IDeplacable** et **IExecutable**. On peut affecter un objet de la classe **TChose** à une variable dont le type est une quelconque de ces classes/interfaces.

15 - Collision de noms

15-1 - Entre une classe et une interface

Chargez le projet interface5.

Dans certains cas d'implémentation d'interface un ou plusieurs noms de méthode de l'interface peuvent entrer en collision avec un nom de méthode de la classe. Dans notre exemple si on ajoute une méthode **Deplace** à notre classe le compilateur ne saura pas s'il s'agit d'une méthode de la classe ou de l'interface implémentée :

```
TChose= Class(TInterfacedObject, IDeplacable)
  //TChose
  Distance : Integer;
  //IDeplacable
  Procedure Deplace;
  Procedure Arret;
  //TChose
  Procedure ProcedureDeTChose;
  Procedure Deplace; //Collision de nom
  Destructor Destroy;Override;
end;
```

La compilation échoue car le compilateur suppose la présence d'une méthode surchargée. On doit donc, pour résoudre ce conflit, indiquer au compilateur que l'implémentation de la méthode **Deplace** de l'interface **IDeplacable** est prise en charge différemment par la classe :

```
TChose= Class(TInterfacedObject, IDeplacable)
  //TChose
  Distance : Integer;
  //IDeplacable
  Procedure IDeplacable.Deplace=DeplaceDeInterface; //Redirection de l'implémentation
  Procedure Arret;
  //TChose
  Procedure ProcedureDeTChose;
  Procedure Deplace;
  Procedure DeplaceDeInterface; //Implémentation réelle de IDeplacable.Deplace
  Destructor Destroy;Override;
end;
```

15-2 - Entre deux interfaces distinctes

Ainsi on peut déclarer 2 méthodes de même nom. Lors de l'appel de **IDeplacable.Deplace**, déclarée dans la classe **TChose**, c'est la méthode *DeplaceDeInterface* qui sera appelée.

Chargez le projet interface51.

Il reste le cas où deux interfaces déclarent chacune un nom de méthode identique :

```
IDeplacable=interface(IInterface)
[ '{8C73D412-47C0-4D42-A492-B6F40433E338}' ]
  Procedure Deplace;
  Procedure Arret;
```

```

End;

IExecutable=interface(IInterface)
[ '{E8CCF73F-A41A-45C9-ABEE-2DD422CD78B4}' ]
  Procedure Demarrage;
  Procedure Arret; //Nom de méthode dupliqué
End;


TChose= Class(TInterfacedObject, IDeplacable, IExecutable)
  //TChose
  Distance : Integer;

  //IDeplacable
  Procedure Deplace;
  Procedure Arret;

  //IExecutable
  Procedure Demarrage;

  //TChose
  Procedure ProcedureDeTChose;
  procedure Stop;
  Destructor Destroy; Override;
end;
    
```

Vous remarquerez que les déclarations précédentes compilent avec succès.

 Dans ce cas l'appel de la méthode **IExecutable.Arret** ne provoque aucune erreur mais surtout ne fait rien.

Ce qui nous oblige à être vigilant et à déclarer la classe comme ceci :

```

TChose= Class(TInterfacedObject, IDeplacable, IExecutable)
  //TChose
  Distance : Integer;

  //IDeplacable
  Procedure Deplace;
  Procedure IDeplacable.Arret=Stop;

  //IExecutable
  Procedure Demarrage;
  Procedure Arret;

  //TChose
  Procedure ProcedureDeTChose;
  procedure Stop;
  Destructor Destroy; Override;
end;
    
```

Le choix des méthodes dupliquées à redéfinir se fait à votre convenance.

15-3 - Entre des interfaces héritées

Chargez le projet interface52.

Dans le cas où la collision se fait sur un arbre d'héritage d'interface, le choix des méthodes dupliquées à redéfinir se fera sur l'interface déclarée :

```
type
  IDeplacable=interface(IInterface)
  ['{8C73D412-47C0-4D42-A492-B6F40433E338}']
  Procedure Deplace;
  Procedure Arret;
End;

IExecutable=interface(IDeplacable)
['{E8CCF73F-A41A-45C9-ABEE-2DD422CD78B4}']
  Procedure Demarrage;
  Procedure Arret;
End;

//Collision de nom de méthode
TChose= Class(TInterfacedObject, IExecutable)
  //TChose
  Distance : Integer;

  //IDeplacable
  Procedure Deplace;
  Procedure IDeplacable.Arret=Stop;

  //IExecutable
  Procedure Demarrage;
  Procedure Arret;

  //TChose
  Procedure ProcedureDeTChose;
  procedure Stop;
  Destructor Destroy;Override;
end;
```

Dans notre exemple la compilation renvoie l'erreur :

```
Identificateur non déclaré : IDeplacable
```

Ce qui nous oblige à redéfinir la méthode **Arret** de l'interface **IExecutable**, seule connue par la classe **TChose**.

16 - Interroger l'arbre d'héritage d'une interface

Chargez le projet interface53.

Dans le dernier exemple nous aimerions savoir en utilisant le code suivant si **IExecutable** supporte (hérite de) **IDeplacable**

:

```
if Supports(MonInterface, IDeplacable) then
begin
  writeln('IExecutable supporte IDeplacable.');
```

Malheureusement ce code renvoie toujours faux. Pourtant notre déclaration d'interface précise bien un héritage :

```
IExecutable=interface(IDeplacable)
```

Revenons sur le problème de collision de nom entre des interfaces héritées, **IDeplacable** ne pouvait être utilisée car inconnue dans la déclaration de la classe :

```
TChose= class(TInterfacedObject, IExecutable)
```

Le compilateur construit la table des interfaces d'une classe uniquement avec les interfaces déclarées dans la classe, si elle n'est pas trouvée on parcourt la table des interfaces des classes ancêtres.

L'ajout de **IDeplacable** dans la liste des interfaces résout notre problème.

```
TChose= class(TInterfacedObject, IDeplacable, IExecutable)
```

Le code suivant devenant possible :

```
var MonInterface : IInterface; //Permet un usage polymorphe
    MonInterfaceHeritee: IDeplacable;

begin
  MonInterface:=TChose.Create;

  if Supports(MonInterface, IDeplacable) then
  begin
    writeln('IExecutable supporte IDeplacable.');
```

```
    MonInterfaceHeritee:=MonInterface as IDeplacable;
    MonInterfaceHeritee.Deplace;
```

```
    MonInterface:=MonInterfaceHeritee;
    IDeplacable(MonInterface).Deplace; // Cast obligatoire
  end;
```

```
  MonInterfaceHeritee:=Nil; //Appel le destructeur
  MonInterface:=Nil; //Interface déjà libérée
  readln;
end.
```

Pour approfondir le sujet je vous conseille la lecture du tutoriel (niveaux confirmé) sur **l'organisation mémoire des classes et des interfaces** par John COLIBRI.

17 - Les interfaces sous Delphi .NET

Sur la plate-forme Win32 seules les classes peuvent implémenter des interfaces, en revanche sous la plate-forme .NET le type enregistrement (**Record**) peuvent également implémenter des interfaces.

Voici ce que donne la transformation de la classe **TChose** en *Record* :

```
TChose= Record(IDeplacable) // Implique de déclarer les méthodes de IInterface
  //TChose
  Distance : Integer;
  //IDeplacable
  Procedure Deplace;
  Procedure Arret;
  //TChose
  Procedure ProcedureDeTChose;
  Constructor Create(ADistance: Integer);
end;
```

Vous trouverez des exemples plus élaborés dans les sources de Delphi .NET:

```
unit Borland.Delphi.System;
...
  TDateTime = packed record(IFormattable, IComparable, IConvertible)
...

```

IFormattable fournit des fonctionnalités permettant de mettre en forme la valeur d'un objet dans une chaîne.

IComparable définit une méthode de comparaison généralisée qu'un type valeur ou qu'une classe implémente pour créer une méthode de comparaison spécifique au type.

IConvertible définit les méthodes qui convertissent la valeur de la référence d'implémentation ou du type valeur en un type Common Language Runtime possédant une valeur équivalente.

18 - Liens

Pour approfondir l'aspect des interfaces notamment dans l'environnement COM consultez ce [tutoriel](#).

Vous y trouverez d'autres informations, par exemple la délégation en utilisant la clause **Implement**, voir aussi [Code Reuse Through Interfaces](#)

Vous pouvez aussi consultez le chapitre 4-12 *Utilisation des interfaces* (page 62) du [Guide du développeur Delphi 7](#).

Vous trouverez un exemple complet d'utilisation des interfaces dans le tutoriel intitulé "[Réaliser un plug-in comportant un composant](#)".

