

# Combiner des procédures et des méthodes

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

Date de publication : 29/08/2007

Dernière mise à jour : 10/10/2007

Il peut être intéressant de pouvoir manipuler indistinctement des procédures et/ou des méthodes afin de partager certains traitements. Voyons comment procéder.  
Je tiens à remercier **Sébastien Doeraene**, **gb\_68** et **Pedro** pour leurs remarques.

---

1 - Public concerné.....	3
1-1 - Les sources.....	3
2 - Rappel sur le type procédure.....	4
3 - Rappel sur le type méthode d'objet.....	6
4 - Le grand mix.....	8
4-1 - Appel d'une méthode en lieu et place d'une procédure.....	8
4-2 - Appel d'une procédure en lieu et place d'une méthode.....	10
5 - Conclusion.....	11
6 - Lien.....	12

## 1 - Public concerné

Débutant	Avancé	Confirmé
<input type="text"/>		

Testé sous Xp et Delphi 2006.

### 1-1 - Les sources

Les fichiers sources des différents exemples :

**FTP.**

**HTTP.**

## 2 - Rappel sur le type procédure

Comme nous le dit la documentation de Delphi :

```
Les types procédure permettent de traiter des procédures et des fonctions comme des valeurs pouvant être affectées à des variables ou transmises à d'autres procédures ou fonctions.
```

```
...
```

```
Deux types de procédure sont compatibles si :
```

- Ils ont la même convention d'appel.
- Ils renvoient le même type de valeur ou pas de valeur.
- Ils ont le même nombre de paramètres, avec le même type aux mêmes positions. Le nom des paramètres est sans importance.

Une variable de type procédure peut également se rapprocher du type pointeur, ces types étant compatibles dans les affectations.

Déclarons un type procédure :

```
type  
  TProcEDURETypee = Procedure(Element : String);
```

On peut dès lors l'utiliser comme paramètre d'une procédure :

```
procedure Enumere(Collection: TStringDynArray ; Traitement:TProcEDURETypee);  
{Enumère une collection de donnée en traitant chaque élément}  
var I : Integer;  
begin  
  for i := Low(Collection) to High(Collection) do  
    Traitement(Collection[i]);  
end;
```

La procédure de traitement, passée en paramètre, doit respecter la signature du type déclaré :

```
Procedure AfficheTableau(Element : String);  
begin  
  Writeln(Element);  
end;
```

Bien évidemment ici cette approche semble suivre le principe du *pourquoi faire simple quand on peut faire compliqué ?* On peut, dans notre cas, envisager des traitements différents lors de l'itération de notre collection, par exemple l'écrire dans un fichier, sur un port série, etc.

L'appel de notre procédure peut se faire directement en utilisant l'opérateur @ :

```
var I : Integer;  
    tabNom : TStringDynArray;  
  
begin  
try  
  //Crée le tableau et le peuple  
  SetLength(tabNom,10+1);  
  For I:=0 to 10 do  
    tabNom[I]:=Chr(I+65);  
  Enumere(tabNom, @AfficheTableau);  
  Readln;  
finally  
  Finalize(tabNom);  
end;
```

Il reste possible d'utiliser une variable :

```
var MaProcedure : TProcedureTypee;  
begin  
  ...  
  maProcedure := @AfficheTableau;  
  Enumere(tabNom, MaProcedure);  
  ...
```

### 3 - Rappel sur le type méthode d'objet

A la différence du type procédural, le type méthode d'objet se déclare en ajoutant le mot clé **of object** :

```
type
  TProcedureObjet = Procedure(Element : String) of object;
```

On remarquera que nos deux déclarations sont très semblables. Déclarons notre classe pour héberger notre méthode **Affiche**:

```
TUneClasse = Class
private
  FProcedure : TProcedureObjet;
public
  Procedure Affiche(Element : String);
  Procedure Iteration(Collection: TStringDynArray);
  property Traitement: TProcedureObjet read FProcedure write FProcedure;
End;
```

Son implémentation étant la suivante :


```
{ TUneClasse }
procedure TUneClasse.Affiche(Element: String);
begin
  Writeln(Element);
end;

procedure TUneClasse.Iteration(Collection: TStringDynArray);
var
  I      : Integer;
begin
  if Assigned(Traitement) then
    for i := Low(Collection) to High(Collection) do
      Traitement(Collection[i]);
end;
```

On utilisera cette méthode d'objet ainsi :

```
UnObjet:= TUneClasse.Create;
UnObjet.Traitement:=UnObjet.Affiche; //Affecte la méthode d'objet.
UnObjet.Iteration(tabNom); //Exécute le traitement
Readln;
```

Notez que l'on peut affecter à la propriété **Traitement** n'importe quelle méthode de toute classe, elle doit juste respecter la signature déclarée dans le type.

 **Attention** cette affectation recèle l'effet de bord suivant :

```
UnObjet:= TUneClasse.Create;
UnAutreObject := TUneAutreClasse.Create;
UnObjet.Traitement:=UnAutreObject.UneMethodeCompatible;
UnAutreObject.Free; // Je n'en ai plus besoin
UnObjet.Iteration(tabNom); // **Problème, la méthode de l'objet référencé n'est plus accessible
```

Si la méthode *UneMethodeCompatible* utilise des variables propres à *UnAutreObject* il ne faut pas le détruire tant que *UnObjet.Traitement* référence une de ces méthodes.

Jusqu'à maintenant ces 2 approches ne diffèrent pas fonctionnellement.

## 4 - Le grand mix

Il peut être intéressant de pouvoir manipuler indistinctement ces 2 types.

### 4-1 - Appel d'une méthode en lieu et place d'une procédure

Essayons d'appeler notre procédure *Enumere* avec une méthode d'objet :


```
Enumere(tabNom, UnObjet.Affiche);
```

Ici la compilation échoue :

```
Erreur E2009 : Types incompatibles : 'procédure normale et pointeur de méthode'
```

Un pointeur de méthode d'objet est en fait une paire de pointeurs, le premier stocke l'adresse d'une méthode et le second une référence à l'objet auquel appartient la méthode. En déclarant le type **TMethod** Delphi autorise la manipulation de ces 2 informations :

```
type
  TMethod = record
    Code,
    Data : Pointer;
  end;
```

 *Sous Win32, les types pointeurs de procédures sont toujours incompatibles avec les types pointeurs de méthodes, mais cela n'est pas vrai sur la plate-forme .NET (puisque tout est objet). La valeur nil peut être affectée à tous les types de procédure.*

Note : Le type **TMethod** existe bien sous Delphi .NET mais est complètement adaptée à cette plate-forme :

```
TMethod = record
public
  var
    Data: TObject;
    Code: TMethodCode; //MemberInfo

    // TODO: We should be able to support ClassVar.Methods and ClassStatic.Methods as well
  constructor Create(AData: TObject; ACode: TMethodCode); overload;
  constructor Create(AData: TObject; const AName: string); overload;
  function Clone: TMethod;

  function CanInvoke: Boolean;
  function Invoke(const AParams: array of TObject): TObject;
  function ToString: string; override;

  function IsEmpty: Boolean;
  class function Empty: TMethod; static;


  class operator Implicit(ADelegate: Delegate): TMethod;
  class operator Equal(const ALeft, ARight: TMethod): Boolean;
  class operator NotEqual(const ALeft, ARight: TMethod): Boolean;
end;
```

Déclarons une variable de type **TMethod** afin de construire notre appel :

```
var MaMethod : TMethod;
```

```
MaProc : TProcedureTypee;
```

On récupère d'abord un pointeur de méthode puis on affecte la partie code à une variable procédurale :

 *Je vous recommande de sauvegarder tous vos travaux avant d'exécuter le code suivant.*

```
MaMethod:=TMethod(UnObjet.Traitement);
MaProc:=MaMethod.Code;
Enumere(tabNom, MaProc);
```

Ce code plante l'application car le compilateur insère dans le code d'appel, et dans celui de l'initialisation de la méthode, la gestion d'un paramètre implicite supplémentaire référant l'adresse de l'objet hébergeant cette méthode. Il manque donc une information sur la pile.

Redéclarons notre type de procédure en prenant en compte ce paramètre supplémentaire :

```
TProcedureTypeeModifiee = Procedure(ParametreVide:Pointer; Element : String);
```

Déclarons une nouvelle procédure **EnumereV2** :

```
procedure EnumereV2(Collection: TStringDynArray ; Traitement:TProcedureTypeeModifiee);
var I : Integer;
begin
  for i := Low(Collection) to High(Collection) do
    Traitement(Nil, Collection[i]);
end;
```

L'ajout d'un paramètre supplémentaire est inutile à nos traitements mais nécessaire pour se *caler* sur le code généré. On peut donc avec cette signature appeler directement notre méthode :

```
EnumereV2(tabNom, @TUneClasse.Affiche);
```

En revanche, et à cause de l'utilisation du nom de classe, on ne peut pas appeler une méthode virtuelle. On utilisera dans ce cas la première approche :

```
var MaProcV2 : TProcedureTypeeModifiee;
...
MaMethod:=TMethod(UnObjet.Traitement); // Peut pointer sur une méthode virtuelle
MaProcV2:=MaMethod.Code;
EnumereV2(tabNom, MaProcV2);
```

Notez qu'à partir du moment on utilise ce type de construction, les propriétés de l'objet ne sont plus accessibles et leurs accès provoqueront un comportement instable de l'application. On peut se protéger de ce comportement en forçant, si cela est possible, le type de la méthode d'objet en méthode de classe :

```
TUneClasse =Class
private
  FProcedure : TProcedureObjet;
public
  class procedure clsAffiche(Element: String);
  ...
```

Dans ce cas l'appel est "simplifié" :

```
EnumereV2(tabNom,@TUneClasse.clsAffiche);
```

## 4-2 - Appel d'une procédure en lieu et place d'une méthode

Pour appeler une procédure via une variable de méthode d'objet on peut utiliser une fonction renvoyant une instance de **TMethod** :

```
Function MakeProcedureOfObject(ProcedureCible : Pointer):TMethod;  
//Construit un record TMethod avec l'adresse d'une procedure classique  
begin  
  Result.code:=ProcedureCible; //L'adresse de la procédure à appeler  
  Result.Data:=Nil; //L'objet est inconnu dans ce cas.  
end;
```

L'appel de la procédure nécessite de transtyper le résultat obtenu :

```
UnObjet.Traitement:=TProcedureObjet(MakeProcedureOfObject(@AfficheTableau2));  
UnObjet.Iteration(tabNom);
```

Cette approche permet par exemple d'affecter à une méthode d'objet une procédure imbriquée. On pourrait très bien renseigner le champ *Data* avec la référence de l'objet mais dans ce cas là autant utiliser directement un objet ;-)

## 5 - Conclusion

Bien évidemment ce que l'on vient de voir ici n'est pas très orthodoxe comme approche mais peut dans certaines circonstances être utile au moins pour la seconde pratique. Je vous laisse seul juge de tels usages.

Sachez enfin que pour la manipulation de hook, ou appel **callback**, Delphi propose la méthode **Classes.MakeObjectInstance** qui se charge de convertir un type particulier de méthode d'objet en un pointeur. Vous pouvez consulter le code source de la VCL ou ce [lien](#) ;-)

## 6 - Lien

Une **autre approche** pour le callback, une autre avec de l'**assembleur** (code à vérifier).

Je vous **recommande** sur le sujet, mais dans une autre catégorie, la lecture de ce tutoriel très pointu nommé **Construire une procédure pointant sur une méthode**, écrit et conçu par Sébastien Doeraene.