

# Partie I : Composants non-visuels

par [Sébastien Doeraene](#)

Date de publication : 07/07/2005

Dernière mise à jour :

Dans cette première partie, vous allez apprendre à réaliser des composants non visuels, c'est-à-dire des composants qui apparaissent sur votre fiche uniquement en conception (comme le TTimer par exemple).

*Ce tutoriel a été écrit avec le support de Delphi 2005 édition Architecte. Si vous possédez une version antérieure ou une édition inférieure, il est possible que vous ne puissiez pas utiliser certaines des fonctionnalités présentées ici.*

*Certains liens dans ce tutoriel font référence à l'aide locale de Delphi 2005 - ils commencent tous par "ms-help://" -, ces liens ne fonctionnent que si vous avez enregistré ce tutoriel sur votre disque local et que vous possédez une version de Delphi supérieure ou égale à la version 2005.*

## Introduction

### I - Qu'est-ce qu'un composant exactement ?

- I-A - Définition
- I-B - Héritage de TPersistent
- I-C - Héritage de TComponent
- I-D - Possession et composition
- I-E - Réutilisation des composants
- I-F - Différences entre développement de composants et d'applications
- I-G - Conclusion

### II - Création d'un composant non-visuel d'exemple

- II-A - Choix du composant à développer
- II-B - Le paquet de nos composants
- II-C - Le code de base de tout composant
- II-D - Sur quoi se base le fonctionnement de notre composant ?
- II-E - Fonctionnement de base
  - II-E-1 - Théorie
  - II-E-2 - Mise en pratique
- II-F - Propriétés dans l'inspecteur d'objet
- II-G - Événements
  - II-G-1 - Théorie
  - II-G-2 - Mise en pratique
- II-H - Valeurs par défaut des propriétés
  - II-H-1 - Comment Delphi enregistre-t-il les fiches
  - II-H-2 - Spécificateurs de stockage
- II-I - Une dernière retouche
- II-J - Code complet de l'unité SelDirDlg

### III - Test du composant en le créant dynamiquement

### IV - Installer le composant dans la palette de Delphi

- IV-A - Un paquet de conception
- IV-B - La procédure Register
- IV-C - La routine RegisterComponents
- IV-D - Recensement du TSelectDirDialog
- IV-E - Code complet de l'unité RegComposTuto
- IV-F - Installation du paquet
- IV-G - Une icône pour le composant

### V - Conclusion

### VI - Liens utiles

### VII - Remerciements

## Introduction

Pour commencer, nous allons découvrir comment créer des composants non-visuels. Ces composants n'apparaissent qu'en mode design sous forme de petit carré avec une icône. Un exemple très utilisé de composant non-visuels est le **TMainMenu**.

Certains se diront, en apprenant que l'on va commencer par les composants non-visuels, que cette partie va être très ennuyeuse et auront envie de passer directement à la suivante.

À ceux-là, je répondrai que les composants non-visuels sont aussi utiles que les composants visuels, et qu'ils facilitent souvent des tâches d'une grande complexité. Vous pouvez prendre à titre d'exemple la bibliothèque **Indy**, dont tous les composants sont non-visuels. Cette bibliothèque est une suite de composants encapsulant des API réseaux.

On s'aperçoit donc que la notion de composant n'est pas liée à la notion de graphisme, puisqu'un composant non-visuel ne *dessine* rien.

Vous voyez que vous ne vous ennuierez pas durant cette partie, et au contraire, vous découvrirez, peut-être avec enthousiasme, les portes que Delphi nous ouvre vers la souplesse de la création de composants.

## I - Qu'est-ce qu'un composant exactement ?

Premièrement, il semble important de faire le point sur ce sujet.

### I-A - Définition

Un *composant*, au sens où on l'entend généralement, est un *objet* (au sens commun du terme, et non au sens de la POO) que l'on peut placer sur une fiche Delphi. En théorie, un composant est un descendant (direct ou non) de la classe de base **TComponent**.

**TComponent** descend de la classe **TPersistent**, elle-même descendante de **TObject**.

### I-B - Héritage de TPersistent

**TPersistent** encapsule le comportement commun à tous les objets pouvant être affectés à d'autres objets et pouvant lire ou écrire leurs propriétés dans un fichier de fiche (fichier .dfm ou .xfm). Cela signifie que les descendants de cette classe seront capables de se *recopier via* la méthode **Assign** et également de *s'enregistrer* dans un flux (*stream*). Bien entendu, pour que ces capacités soient supportées, il faut implémenter certaines choses.

### I-C - Héritage de TComponent

La classe **TComponent** est l'ancêtre partagé de chaque composant de la VCL. **TComponent** implémente directement l'une de ces capacités, à savoir le fait qu'il peut être enregistré dans un flux. Nous pouvons remercier Borland de cela, car sinon nous devrions effectuer un travail considérable à chaque fois que l'on créerait un nouveau composant !

Ainsi, tout composant enregistre automatiquement toutes ses propriétés déclarées **published** (nous verrons plus loin ce que cela signifie ; pour l'instant, contentez-vous de savoir que c'est une directive au même titre que **public**).

### I-D - Possession et composition

**TComponent** apporte également d'autres fonctionnalités, telles que le fait que tout descendant de **TComponent** possède un propriétaire (*owner*) qui se chargera de le détruire lorsque lui-même sera détruit.

On peut effectuer ici un parallèle avec les Lego : il existe des *blocs* de base avec lesquels on peut en créer de plus complexes.

*On retrouve ici la notion de **composition** de UML. Le propriétaire désigne la relation particulière entre un composant maître et plusieurs objets le constituant. Le propriétaire étant responsable de la libération des parties qui le composent. Les composants ont donc une existence dépendante du composé.*

*Exemple : Une fenêtre Windows est composée de menus, d'ascenseurs, de boutons... Si la fenêtre est détruite alors tous les composants (menus, ascenseurs, boutons...) sont également détruits.*

Chaque objet peut être propriétaire de 0 ou n autres objets. Par contre, il est impossible de créer un *cycle* de possession : si, à partir d'un objet donné, on remonte chaque fois à son propriétaire, on ne pourra jamais retrouver le premier objet.



Sachez, pour en terminer à propos de la possession, que tout composant placé sur une fiche a pour propriétaire cette fiche.

## I-E - Réutilisation des composants

Le principal atout de la POO est la *réutilisation du code*.

Les composants n'échappent pas à cette règle, aussi peut-on, à partir d'un composant, l'améliorer facilement en dérivant un nouveau composant de celui-ci.

## I-F - Différences entre développement de composants et d'applications

*Extrait de la documentation de Delphi 6 sur les classes et composants*

Comme les composants sont des classes, les créateurs de composants manipulent les objets à un niveau différent de celui des développeurs d'applications. La création de nouveaux composants nécessite de dériver de nouvelles classes.

Brièvement, il existe deux différences principales entre la création des composants et leur utilisation dans des applications. Pour la création de composants, vous avez accès à des parties de la classe qui sont inaccessibles aux programmeurs d'applications. Vous ajoutez de nouvelles parties (des propriétés, par exemple) aux composants.

A cause de ces différences, il faut prendre en compte un plus grand nombre de conventions, et réfléchir à la manière dont les développeurs d'applications vont utiliser les composants.

## I-G - Conclusion

Bref, les composants permettent à tous les développeurs d'augmenter les possibilités de Delphi, et donc de simplifier le développement d'applications.

Vous savez désormais ce qu'est un composant au sens premier du terme. Nous allons maintenant entrer dans le vif du sujet : la création de votre premier composant !

## II - Création d'un composant non-visuel d'exemple

Le meilleur moyen d'apprendre à faire quelque chose, c'est de le faire ! Donc nous allons voir pas à pas comment créer un composant non-visuel.

### II-A - Choix du composant à développer

J'ai décidé de réaliser avec vous un composant du nom de **TSelectDirDialog**.

Vous devriez connaître le composant **TOpenDialog**, qui permet de sélectionner un fichier à ouvrir. Notre **TSelectDirDialog** fera plus ou moins de même mais permettra de sélectionner un dossier plutôt qu'un fichier.

Bien entendu nous ne dessinerons pas la fenêtre correspondante : nous utiliserons les APIs de Windows pour utiliser cette boîte classique, tout comme le fait le composant **TOpenDialog**.

Ce composant permettra donc d'aborder les points suivants :

- Les propriétés affichées dans l'inspecteur d'objet
- La création d'événements
- L'encapsulation des APIs Windows par un composant
- L'installation du composant dans la palette de Delphi
- L'association d'une icône au composant

Voilà un beau programme non ? Alors allons-y !

### II-B - Le paquet de nos composants

Dans les deux premières versions de Delphi, les composants étaient tous compilés dans une seule et même librairie appelée **CmpLib32.bpl**. Cette DLL contenait le code de tous les composants installés et il était difficile d'en télécharger.

Delphi 3 a apporté les **paquets**. Désormais, tous les composants seraient compilés dans des paquets. Cela a permis de simplifier énormément le chargement des composants utiles, ce qui devenait presque indispensable au vu des nombreux composants disponibles sur Internet (notamment l'incontournable JVCL).

Tout au long de ce cours, nous développerons quatre composants. Nous les compilerons tous les quatre dans un seul et même paquet. Nous allons donc commencer par créer ce paquet.

Créez un nouveau paquet, et nommez-le **ComposTutoR.bpl**. Vous pouvez bien sûr le nommer comme bon vous semble, mais sachez que c'est comme cela qu'il sera dénommé dans la suite de ce cours. Enregistrez-le dans un dossier dédié, par exemple sous le dossier **<Delphi>\source\ComposTuto\**.

Sélectionnez le menu **Projet|Options** pour afficher les options du paquet. Dans **Description**, vous pouvez indiquer une description du paquet mais surtout vous pouvez spécifier qu'il s'agit d'un paquet de type *runtime* (Options d'utilisation > Seulement en exécution).

*Le **R** que l'on ajoute à la fin du nom du paquet permet de nous rappeler (à nous et aux développeurs qui utiliseront le paquet) que c'est un paquet de type runtime. Cela veut dire que ce paquet ne peut pas être installé dans l'EDI de Delphi. Le paquet qui sera installé (et qui se nommera **ComposTutoD.bpl**, avec **D** pour design) ne contiendra pas de code utile au fonctionnement du composant.*

*Il est aussi possible de créer des paquets de type runtime et design (des **MonPaquetRD.bpl** si vous voulez) mais un paquet ne devrait normalement jamais être de ce type : les paquets de type design ne doivent contenir que du code servant à l'intégration dans l'EDI de Delphi, et ne servent donc à rien en dehors.*

## II-C - Le code de base de tout composant

Créez une nouvelle unité, nommée **SelDirDlg.pas**, dans notre paquet **ComposTutor.bpl**. Cette unité contiendra le code de notre composant **TSelectDirDialog**.

Un composant non-visuel descend de **TComponent**, et éventuellement de l'un de ses descendants. Dans notre cas, il descendra directement de **TComponent**.

Ainsi, le code minimal d'une unité d'un composant est celui-ci :

```
unit SelDirDlg;

interface

uses
  Classes;

type
  TSelectDirDialog = class(TComponent)
  end;

implementation

end.
```

Notez que nous utilisons l'unité **Classes**, qui déclare les classes **TPersistent** et **TComponent** (entre autres).

## II-D - Sur quoi se base le fonctionnement de notre composant ?

### II-E - Fonctionnement de base

Commençons par faire fonctionner notre composant avec ses propriétés de base. Ensuite, nous l'améliorerons avec quelques techniques indispensables à la réalisation de composants dignes de ce nom.

#### II-E-1 - Théorie

Tout d'abord, comme nous allons beaucoup utiliser des chaînes de caractères représentant des noms de dossiers, il serait pratique de disposer d'un type prévu pour. Nous allons le déclarer comme le type **TFileName** qui, je vous le rappelle, désigne une chaîne de caractères représentant un nom de fichier :

```
type
  TDirName = type string;
```

Il faut aussi savoir comment nous comptons utiliser la boîte de dialogue classique de Windows...

L'unité **FileCtrl** propose une routine **SelectDirectory** surchargée de deux façons :

```
function SelectDirectory(var Directory: string;
  Options: TSelectDirOpts; HelpCtx: Longint): Boolean; overload;
function SelectDirectory(const Caption: string; const Root: WideString;
```

```
var Directory: string; Options: TSelectDirExtOpts = [sdNewUI]; Parent: TWinControl = nil): Boolean; overload;
```

Ces deux surcharges font apparaître la boîte de dialogue avec diverses informations.

Nous utiliserons la seconde surcharge, car elle est plus complète.

Définissons maintenant quelques propriétés pour notre composant. Nous aurons besoin des divers renseignements demandés en paramètres par la fonction **SelectDirectory**.

## II-E-2 - Mise en pratique

Nous aurons donc besoin de quatre propriétés :

Utilisation	Propriété	Type
Titre de la boîte de dialogue	<b>Caption</b>	string
Répertoire racine	<b>Root</b>	TDirName
Répertoire sélectionné	<b>Directory</b>	TDirName
Options de la boîte de dialogue	<b>Options</b>	TSelectDirExtOpts

Nous ajouterons un constructeur pour les initialiser et une méthode **Execute** qui affichera la boîte de dialogue :

```
function TSelectDirDialog.Execute : boolean;
begin
  Result := SelectDirectory(FCaption, FRoot, string(FDirectory), FOptions);
end;
```

Rappelons la déclaration de la fonction **SelectDirectory** afin de comprendre les différents paramètres passés :

```
function SelectDirectory(const Caption: string; const Root: WideString;
var Directory: string; Options: TSelectDirExtOpts = [sdNewUI]; Parent: TWinControl = nil): Boolean; overload;
```

Rien de tout ceci n'est plus compliqué que la création d'une classe implémentant une certaine fonctionnalité. Nous ne nous attarderons pas là-dessus.

Voici le code actuel :

```
unit SelDirDlg;

interface

uses
  Classes, FileCtrl;

type
  TDirName = type string;

  TSelectDirDialog = class(TComponent)
  private
    FCaption : string;
    FRoot : TDirName;
    FDirectory : TDirName;
    FOptions : TSelectDirExtOpts;
  public
    constructor Create(AOwner : TComponent); override;

    function Execute : boolean;
```



```

property Caption : string read FCaption write FCaption;
property Root : TDirName read FRoot write FRoot;
property Directory : TDirName read FDirectory write FDirectory;
property Options : TSelectDirExtOpts read FOptions write FOptions;
end;

implementation

constructor TSelectDirDialog.Create(AOwner : TComponent);
begin
  inherited;
  FCaption := 'Sélectionnez un dossier';
  FRoot := 'C:\';
  FDirectory := 'C:\';
  FOptions := [sdNewUI];
end;

function TSelectDirDialog.Execute : boolean;
begin
  Result := SelectDirectory(FCaption, FRoot, string(FDirectory), FOptions);
end;

end.

```

## II-F - Propriétés dans l'inspecteur d'objet

Jusqu'à présent, nous n'avons fait qu'écrire une simple classe. Les nouveautés vont maintenant commencer.

Une des qualités essentielles des composants est que leurs propriétés sont éditables en mode conception. Or, bien que la classe **TSelectDirDialog** soit un composant, ses propriétés n'apparaîtront pas dans l'inspecteur d'objet avec ce code.

Nous pouvons voir ici une des merveilles de la technologie orientée composants de Borland : il suffit de déclarer les propriétés que l'on souhaite voir apparaître dans l'inspecteur comme **published** (qui se traduit par *publié*).

**published** est une directive de visibilité au même titre que **public** ou **private**. C'est la visibilité la plus grande : elle a les mêmes propriétés que la visibilité **public** mais elle permet aussi de conserver les RTTI (**RunTime Type Information** : Informations de type à l'exécution) qui permettent (notamment) à l'inspecteur d'objet de les *voir* et de les afficher.

```

published
property Caption : string read FCaption write FCaption;
property Root : TDirName read FRoot write FRoot;
property Directory : TDirName read FDirectory write FDirectory;
property Options : TSelectDirExtOpts read FOptions write FOptions;
end;

```

## II-G - Événements

Que seraient les composants sans les événements ? C'est le coeur même de la programmation événementielle. Voyons donc comment ajouter des événements à vos composants.

Toutefois, vous pouvez remarquer que notre composant fonctionne très bien sans événement. Les événements ne sont pas indispensables à la création de composants.

### II-G-1 - Théorie

En Delphi, il est très simple d'ajouter un événement à vos composants. Voici comment procéder.



Premièrement, vous avez besoin d'un type *événementiel*. En réalité, ce sont des types *méthodes*.

Un type méthode est déclaré de cette façon :

```
type
  TTypeMethode = procedure(Param1 : TType1; Param2 : TType2; ...) of object;
```

Les deux mots-clés **of object** signifient qu'il s'agit d'un type **méthode** et non d'un type **routine**. N'oublions pas qu'une méthode est une routine de classe.

Dans le cas d'un type événementiel, un type méthode est requis, et donc ces deux mots-clés le sont aussi.

On peut déclarer des variables de ce type de la même manière que n'importe quel autre.

Pour les utiliser dans la partie gauche d'une affectation, il suffit d'utiliser le nom de la variable à gauche et le nom de la fonction dont on veut enregistrer l'adresse à droite (il est également possible d'affecter la valeur **nil** qui permettra de tester si la variable est assignée ou non) :

```
var VarMethode : TTypeMethode;
...
VarMethode := nil;
VarMethode := MaMethode;
```

Pour appeler la méthode associée, il suffit d'utiliser la variable comme s'il s'agissait du nom de la méthode en question :

```
VarMethode(Param1, Param2, ...);
```

Si l'on veut tester qu'une méthode a été assignée, on peut utiliser la fonction **Assigned** qui prend pour unique paramètre la variable méthode et renvoie **True** si et seulement si elle n'est pas vide (différent de **nil**).

Finalement, mais cela est moins important et uniquement à titre indicatif, si vous devez comparer deux variables de type méthode pour savoir si elles contiennent la même adresse, vous devrez les transtyper en type **TMethod** et comparer les champs **Code** et **Data** de ce **record**.

```
if (TMethod(VarMethode).Code = TMethod(VarMethode2).Code) and // C'est la même méthode
    (TMethod(VarMethode).Data = TMethod(VarMethode2).Data) then // C'est le même objet
...

```

## II-G-2 - Mise en pratique

Nous allons donc ajouter deux événements à notre composant : un qui sera appelé lorsque l'utilisateur aura cliqué sur OK dans la boîte de dialogue, et un autre lorsqu'il aura cliqué sur Annuler.

Le second n'est qu'un événement de *notification*, donc qui n'a pas besoin de paramètre particulier en dehors du paramètre **Sender** qui indique le composant appelant l'événement. Nous utiliserons donc pour celui-là le type **TNotifyEvent**, déclaré comme suit dans l'unité **Classes** :

```
type
  TNotifyEvent = procedure(Sender : TObject) of object;
```

Pour le premier, nous aurons besoin d'un type d'événement spécifique puisqu'il devra envoyer comme paramètre

le dossier sélectionné dans la boîte de dialogue. Voici la déclaration de ce dernier :

```
type
  TAcceptDirEvent = procedure(Sender : TObject; Directory : TDirName) of object;
```

Voici comment se présente maintenant notre interface :

```
interface
uses
  Classes, FileCtrl;

type
  TDirName = type string;
  TAcceptDirEvent = procedure(Sender : TObject; Directory : TDirName) of object;

  TSelectDirDialog = class(TComponent)
  private
    FCaption : string;
    FRoot : TDirName;
    FDirectory : TDirName;
    FOptions : TSelectDirExtOpts;

    FOnAccept : TAcceptDirEvent;
    FOnCancel : TNotifyEvent;
  public
    constructor Create(AOwner : TComponent); override;

    function Execute : boolean;
  published
    property Caption : string read FCaption write FCaption;
    property Root : TDirName read FRoot write FRoot;
    property Directory : TDirName read FDirectory write FDirectory;
    property Options : TSelectDirExtOpts read FOptions write FOptions;

    property OnAccept : TAcceptDirEvent read FOnAccept write FOnAccept;
    property OnCancel : TNotifyEvent read FOnCancel write FOnCancel;
  end;
```

Il ne reste plus qu'à appeler ces événements lorsque le cas se présente. Rappelons qu'il suffit d'utiliser le nom de la variable comme nom de la méthode à appeler.

```
function TSelectDirDialog.Execute : boolean;
begin
  Result := SelectDirectory(FCaption, FRoot, string(FDirectory), FOptions);
  if Result then
  begin
    if Assigned(FOnAccept) then
      FOnAccept(Self, FDirectory);
  end else
  begin
    if Assigned(FOnCancel) then
      FOnCancel(Self);
  end;
end;
```

## II-H - Valeurs par défaut des propriétés

Il reste une légère imperfection à notre composant. Et même des développeurs expérimentés dans la réalisation de composants oublient parfois d'ajouter ce détail.

Mais avant de corriger l'imperfection, il faudrait savoir ce qui cloche...

### II-H-1 - Comment Delphi enregistre-t-il les fiches

Vous êtes-vous jamais demandé comment Delphi enregistrerait tout ce que vous lui indiquiez lorsque vous êtes en mode conception. Il est assez simple de le savoir en cliquant droit sur une fiche en mode conception et sélectionnez le menu **Voir comme texte**.

Votre fiche disparaîtra alors pour laisser place à sa forme *texte*. Voici un exemple avec un bouton et un éditeur :

```
object FormMain: TFormMain
  Left = 0
  Top = 0
  Width = 434
  Height = 320
  Caption = 'Essais'
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  OnCreate = FormCreate
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 32
    Top = 32
    Width = 121
    Height = 33
    Caption = 'Button1'
    TabOrder = 0
  end
  object Edit1: TEdit
    Left = 32
    Top = 96
    Width = 361
    Height = 21
    TabOrder = 1
    Text = 'Edit1'
  end
end
```

## II-H-2 - Spécificateurs de stockage

Mais qu'y a-t-il donc d'intéressant à découvrir dans ce code ? Eh bien, c'est que toutes les propriétés ne sont pas enregistrées ! Si vous consultez l'inspecteur d'objet pour ces trois types de composants, vous remarquerez qu'il y a beaucoup plus de propriétés que ça.

Ceci s'explique par le fait que Delphi n'enregistre pas les propriétés dont la valeur est celle par défaut. Mais comment sait-il quelle est la valeur par défaut d'une propriété ? C'est ce que nous allons voir.

Dans l'interface d'une classe de composant (et même de persistant, donc descendant de **TPersistent**), on peut définir les valeurs par défaut d'une propriété au moyen des spécificateurs **default** ou **stored**. On peut aussi utiliser le spécificateur **nodefault** pour supprimer une valeur par défaut héritée. Voici comment utiliser ces spécificateurs.

Pour les propriétés de type scalaire, il suffit d'utiliser le spécificateur **default** suivi de la valeur par défaut. Voici deux exemples :

```
// Propriétés scalaires avec valeur par défaut
type
  TMonSet = set of #0..#31;
  ...

property MonEntier : integer read FMonEntier write FMonEntier default 1;
property MonSet : TMonSet read FMonSet write FMonSet default [#0, #10, #13];
```

Signalons à cette occasion qu'il est impossible d'utiliser des propriétés publiées de type ensemble dont la taille est plus grande que 4 octets, donc qui contient plus de 32 éléments (d'où la définition `TMonSet = set of #0..#31`).

Une valeur par défaut ne sert qu'à savoir s'il faut enregistrer la variable ou pas. À la relecture du fichier `.dfm`, cette valeur par défaut n'est pas utilisée. Vous devez donc prendre garde à bien initialiser la propriété dans le constructeur de l'objet.

Pour les valeurs chaînes, cela se complique un peu. Si la valeur par défaut est une chaîne vide, ne touchez à rien, Delphi utilise cette valeur comme valeur par défaut si rien n'est indiqué.

Si vous voulez utiliser une autre valeur par défaut, vous devrez définir un spécificateur **stored**. Ce qui suit ce spécificateur est un nom de méthode de la classe en cours qui ne prend aucun paramètre et qui renvoie une valeur de type **boolean**. Cette valeur doit être **True** s'il faut enregistrer la valeur et **False** dans le cas contraire. Voici un exemple :

```
// Propriétés non scalaires avec valeur par défaut
function MaChaineStored : boolean;
property MaChaine : string read FMaChaine write FMaChaine stored MaChaineStored;

...

function TMonObjet.MaChaineStored : boolean;
begin
  Result := MaChaine <> 'C:\Temp\';
end;
```

Si vous voulez supprimer toute valeur par défaut, vous pouvez utiliser le spécificateur **nodefault** :

```
property Font nodefault;
```

Vous pouvez obtenir le même résultat en utilisant **stored True**, puisque cette dernière spécification indique que la propriété sera enregistrée si **True** est vrai, donc toujours. De même, vous pouvez utiliser **stored False** si vous voulez que la propriété ne soit jamais enregistrée.

Code extrait de l'aide de Delphi 6 :

```
type
  TSampleComponent = class(TComponent)
  protected
    function StoreIt: Boolean;
  public
    ...
  published
    property Important: Integer stored True; { toujours stockée }
    property Unimportant: Integer stored False; { jamais stockée }
    property Sometimes: Integer stored StoreIt; { dépend de la valeur de la fonction }
  end;
```

Vous pouvez aussi redéfinir la valeur par défaut d'une propriété héritée très simplement :

```
property Visible default False;
property Font stored FontStored;
property Color nodefault;
```

Voici un petit récapitulatif :

Pour...	Indiquer...
Enregistrer si la valeur est différente d'une valeur par défaut (type scalaire)	<b>default</b> ValeurParDefaut
Toujours enregistrer la valeur	rien ou <b>nodefault</b> ou <b>stored</b> True
Ne jamais enregistrer la valeur	<b>stored</b> False
Enregistrer la valeur si la valeur de retour d'une fonction est True	<b>stored</b> NomFonction

Si toutefois vous n'êtes pas certain de ce que vous faites, vérifiez que tout se passe bien dans le fichier .dfm sous forme texte.

Il ne reste plus qu'à mettre cela en pratique. Avec toute la théorie que nous avons acquise, je crois qu'il n'y a plus besoin de détailler la procédure.

## II-I - Une dernière retouche

Nous allons apporter une dernière retouche à notre composant. Que se passerait-il en effet si le disque par défaut du système sur lequel tournait l'application n'était pas C:\ ?

On ne sait en effet jamais dans quel contexte sera utilisé le composant, surtout si on compte le distribuer sur Internet. Il n'est donc pas bon de faire des suppositions quant à l'utilisation de ses composants.

En réalité, dans ce cas, il ne se passerait pas grand chose, car la routine **SelectDirectory** teste les paramètres pour vérifier qu'ils sont valides, et les ignore sinon.

Pourtant, cela fait peu *professionnel*. Voyons donc comment remédier à ce désagrément.

Le coeur de la solution réside dans la routine [GetWindowsDirectory](#).

Malheureusement, cette routine fait partie de l'unité **Windows**, car c'est une API, et utilise donc des paramètres peu confortables (buffer de type PChar).

Déclarons une variable **MainDisk** dans la partie implémentation de l'unité :

```
var
  MainDisk : TDirName;
```

Aux quatre endroits où nous avons utilisé C:\, nous allons utiliser cette variable à la place.

Nous déterminerons cette valeur dans la partie **initialization** de l'unité. Voici comment procéder :

```
initialization
  SetLength(MainDisk, MAX_PATH);
  SetLength(MainDisk, GetWindowsDirectory(PChar(MainDisk), MAX_PATH));
  MainDisk := IncludeTrailingPathDelimiter(ExtractFileDrive(MainDisk));
end.
```

Nous aurons pris soin de rajouter les unités **Windows** et **SysUtils** dans les **uses**.

Après avoir récupéré le dossier de Windows, nous en extrayons le disque et nous nous assurons que le séparateur de dossier (donc le \ sous Windows) est bien ajouté en fin de chaîne.

## II-J - Code complet de l'unité SelDirDlg

Voici le code complet et définitif de l'unité **SelDirDlg** :

### SelDirDlg.pas

```

unit SelDirDlg;

interface

uses
  Windows, SysUtils, Classes, FileCtrl;

type
  TDirName = type string;
  TAcceptDirEvent = procedure(Sender : TObject; Directory : TDirName) of object;

  TSelectDirDialog = class(TComponent)
  private
    FCaption : string;
    FRoot : TDirName;
    FDirectory : TDirName;
    FOptions : TSelectDirExtOpts;

    FOnAccept : TAcceptDirEvent;
    FOnCancel : TNotifyEvent;

    // Ces trois méthodes servent à indiquer si les propriétés Caption, Root et Dir
    // doivent être enregistrée dans le fichier .dfm
    function CaptionStored : boolean;
    function RootStored : boolean;
    function DirStored : boolean;
  public
    constructor Create(AOwner : TComponent); override;

    // Affiche la boîte de dialogue ; renvoie True si l'utilisateur clique sur OK
    function Execute : boolean;
  published
    // Titre de la boîte de dialogue
    property Caption : string read FCaption write FCaption stored CaptionStored;
    // Répertoire racine de l'arbre dans la boîte de dialogue
    property Root : TDirName read FRoot write FRoot stored RootStored;
    // Répertoire sélectionné (avant et après) dans la boîte de dialogue
    property Directory : TDirName read FDirectory write FDirectory stored DirStored;
    // Diverses options pour la boîte de dialogue
    property Options : TSelectDirExtOpts read FOptions write FOptions default [sdNewUI];

    // Déclenché lorsque l'utilisateur clique sur OK dans la boîte de dialogue
    property OnAccept : TAcceptDirEvent read FOnAccept write FOnAccept;
    // Déclenché lorsque l'utilisateur clique sur Annuler dans la boîte de dialogue
    property OnCancel : TNotifyEvent read FOnCancel write FOnCancel;
  end;

implementation

var
  MainDisk : TDirName;

constructor TSelectDirDialog.Create(AOwner : TComponent);
begin
  inherited;
  FCaption := 'Sélectionnez un dossier';
  FRoot := MainDisk;
  FDirectory := MainDisk;
  FOptions := [sdNewUI];
  FOnAccept := nil;
  FOnCancel := nil;
end;

```

## SelDirDlg.pas

```
function TSelectDirDialog.CaptionStored : boolean;
begin
  Result := FCaption <> 'Sélectionnez un dossier';
end;

function TSelectDirDialog.RootStored : boolean;
begin
  Result := FRoot <> MainDisk;
end;

function TSelectDirDialog.DirStored : boolean;
begin
  Result := FDirectory <> MainDisk;
end;

function TSelectDirDialog.Execute : boolean;
begin
  Result := SelectDirectory(FCaption, FRoot, string(FDirectory), FOptions);
  if Result then
  begin
    if Assigned(FOnAccept) then
      FOnAccept(Self, FDirectory);
    end else
    begin
      if Assigned(FOnCancel) then
        FOnCancel(Self);
      end;
    end;
  end;

initialization
  SetLength(MainDisk, MAX_PATH);
  SetLength(MainDisk, GetWindowsDirectory(PChar(MainDisk), MAX_PATH));
  MainDisk := IncludeTrailingPathDelimiter(ExtractFileDrive(MainDisk));
end.
```

Nous allons maintenant tester notre composant.

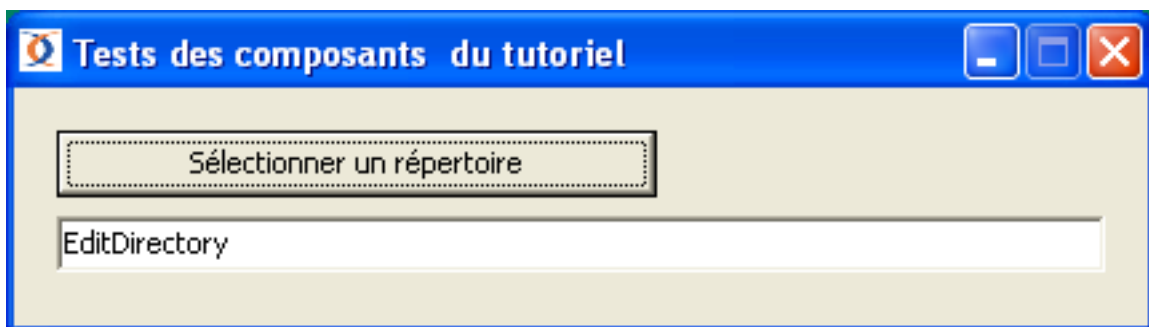


### III - Test du composant en le créant dynamiquement

Pour tester un composant, il est plus prudent et souvent plus facile de le créer dynamiquement sur une fiche et de manipuler ses propriétés à l'exécution.

Créez donc un nouveau projet. Nous le nommerons **TestComposTuto.exe**. Pour faciliter les choses, nous allons créer un groupe de projets **ComposTuto.bpg** (D7 et inférieurs) ou **ComposTuto.bdsgroup** (D2005 et supérieurs). Nous pouvons déjà y mettre le paquet **ComposTutoR** et le projet **TestComposTuto**. Nous appellerons l'unité principale **TestComposTutoMain.pas**.

Nommez la fiche **FormMain**, ensuite placez-y un bouton et un éditeur. Nommez le bouton **ButtonSelectDir** et l'éditeur **EditDirectory** (c'est mon choix, faites ce que vous voulez). Vous obtiendrez ceci :



Dans la déclaration de la classe, dans la partie **public**, déclarez le composant **SelectDirDialog** de type **TSelectDirDialog**. N'oubliez pas d'ajouter l'unité **SelDirDlg** dans les **uses**.

Dans l'événement **OnCreate** de la fiche, nous allons créer le composant et lui assigner quelques propriétés. Nous en profiterons pour placer le dossier de l'application dans l'éditeur.

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  SelectDirDialog := TSelectDirDialog.Create(Self);
  SelectDirDialog.Caption := 'Vous pouvez choisir un dossier';
  EditDirectory.Text := ExtractFilePath(Application.ExeName);
end;
```

Nous avons choisi **Self** comme propriétaire du composant afin de nous décharger de sa destruction finale.

Dans l'événement **OnClick** du bouton, nous ferons apparaître la boîte de dialogue, en ayant pris soin de mettre comme dossier par défaut le dossier écrit dans l'éditeur.

```
procedure TFormMain.ButtonSelectDirClick(Sender: TObject);
begin
  SelectDirDialog.Directory := EditDirectory.Text;
  SelectDirDialog.Execute;
end;
```

Finalement, pour mettre à jour le contenu de l'éditeur lorsque l'utilisateur sélectionne un fichier, nous utiliserons un gestionnaire d'événement écrit à *la main* (n'oubliez pas de le déclarer également dans la partie **public** de la déclaration de la classe) :

```
procedure TFormMain.SelectDirDialogAccept(Sender: TObject; Directory : TDirName);
begin
  EditDirectory.Text := Directory;
end;
```

Nous devons aussi affecter ce gestionnaire à l'événement, dans le **OnCreate** de la fiche :

```
SelectDirDialog.OnAccept := SelectDirDialogAccept;
```

Voici le code complet actuel de l'unité **TestComposTutoMain.pas** :

```
TestComposTutoMain.pas
unit TestComposTutoMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, SelDirDlg;

type
  TFormMain = class(TForm)
    ButtonSelectDir: TButton;
    EditDirectory: TEdit;
    procedure FormCreate(Sender: TObject);
    procedure ButtonSelectDirClick(Sender: TObject);
  private
    { Déclarations privées }
  public
    { Déclarations publiques }
    SelectDirDialog: TSelectDirDialog;
    procedure SelectDirDialogAccept(Sender: TObject; Directory : TDirName);
  end;

var
  FormMain: TFormMain;

implementation

{$R *.dfm}

procedure TFormMain.FormCreate(Sender: TObject);
begin
  SelectDirDialog := TSelectDirDialog.Create(Self);
  SelectDirDialog.Caption := 'Vous pouvez choisir un dossier';
  SelectDirDialog.OnAccept := SelectDirDialogAccept;
  EditDirectory.Text := ExtractFilePath(Application.ExeName);
end;

procedure TFormMain.ButtonSelectDirClick(Sender: TObject);
begin
  SelectDirDialog.Directory := EditDirectory.Text;
  SelectDirDialog.Execute;
end;

procedure TFormMain.SelectDirDialogAccept(Sender: TObject; Directory : TDirName);
begin
  EditDirectory.Text := Directory;
end;

end.
```

Nous pouvons maintenant lancer le programme. Vous pouvez cliquer sur le bouton pour afficher la boîte de dialogue et, lorsque vous cliquez sur OK, le contenu de l'éditeur est mis à jour. Notez qu'il est évidemment possible de modifier directement le chemin dans l'éditeur ; pour y remédier (si vous le voulez), il suffit de passer sa propriété **ReadOnly** à **True**.

Maintenant que nous savons que notre composant fonctionne, nous pouvons l'installer dans la palette des composants afin de pouvoir le placer facilement sur n'importe quelle fiche.

## IV - Installer le composant dans la palette de Delphi

Il reste une dernière étape dans la réalisation de notre composant : l'installer dans la palette des composants de Delphi.

Cette opération, qui peut sembler fastidieuse, est en fait très simple à faire. Nous allons voir pas à pas comment.

### IV-A - Un paquet de conception

Pour pouvoir installer des composants dans la palette des composants, il faut réaliser un paquet de conception. Créez donc, dans le groupe de projets, un paquet nommé **ComposTutoD.bpl**.

Sélectionnez le menu **Projet|Options**. Saisissez une description pour le paquet et choisissez un type de paquet de conception.

Ajoutez une nouvelle unité que nous nommerons **RegComposTuto**. J'utilise **Reg** pour me rappeler que ce type d'unité ne fait rien d'autre que d'*enregistrer (register)* des composants dans la palette.

### IV-B - La procédure Register

Dans cette unité, il faut déclarer une procédure nommée **Register** (elle **doit** se nommer comme cela) sans paramètre :

```
unit RegComposTuto;
interface
procedure Register;
implementation
procedure Register;
begin
end;
end.
```

Cette procédure est reconnue de manière spéciale par l'EDI de Delphi. Vous devrez placer tout le code devant enregistrer les composants dans cette procédure.

*Il ne faut jamais appeler vous-même cette procédure !*

### IV-C - La routine RegisterComponents

La routine que nous utiliserons pour recenser (ou enregistrer) le composant sera la routine **RegisterComponents** de l'unité **Classes**. Il faudra donc ajouter l'unité **Classes** dans les **uses**. Voici la déclaration de cette routine :

```
procedure RegisterComponents(const Page: string;
const ComponentClasses: array of TComponentClass);
```

Le paramètre **Page** est de type chaîne et indique le nom de l'onglet (ou du groupe de boutons pour D2005) dans lequel seront placés les composants.



Le paramètre **TComponentClasses** est un tableau ouvert de classes de composants.

#### IV-D - Recensement du TSelectDirDialog

Pour recenser le composant **TSelectDirDialog** dans l'onglet **Tutoriel**, nous appellerons donc **RegisterComponents** comme suit :

```
RegisterComponents('Tutoriel', [TSelectDirDialog]);
```

N'oubliez pas que pour pouvoir utiliser le type **TSelectDirDialog**, vous devrez rajouter l'unité **SelDirDlg** dans les **uses**.

#### IV-E - Code complet de l'unité RegComposTuto

Voici donc le code complet de l'unité **RegComposTuto** :

```
unit RegComposTuto;

interface

uses
  Classes, SelDirDlg;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Tutoriel', [TSelectDirDialog]);
end;

end.
```

#### IV-F - Installation du paquet

Il ne reste plus qu'à compiler le package et à l'installer. Pour installer le package, cliquez sur le bouton **Installer** (D7 et inférieurs) ou cliquez droit sur le paquet et sélectionnez **Installer** (D2005 et supérieurs).

Vous pouvez désormais rouvrir le projet **TestComposTuto**, et placer le composant directement sur la fiche. Voici ce que donne le code final de la fiche du projet :

##### TestComposTutoMain.pas

```
unit TestComposTutoMain;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, SelDirDlg;

type
  TFormMain = class(TForm)
    ButtonSelectDir: TButton;
    EditDirectory: TEdit;
    SelectDirDialog: TSelectDirDialog;
    procedure FormCreate(Sender: TObject);
    procedure ButtonSelectDirClick(Sender: TObject);
    procedure SelectDirDialogAccept(Sender: TObject; Directory : TDirName);
  private
    { Déclarations privées }
  end;

end.
```

## TestComposTutoMain.pas

```

public
  { Déclarations publiques }
end;

var
  FormMain: TFormMain;

implementation
  {$R *.dfm}

procedure TFormMain.FormCreate(Sender: TObject);
begin
  EditDirectory.Text := ExtractFilePath(Application.ExeName);
end;

procedure TFormMain.ButtonSelectDirClick(Sender: TObject);
begin
  SelectDirDialog.Directory := EditDirectory.Text;
  SelectDirDialog.Execute;
end;

procedure TFormMain.SelectDirDialogAccept(Sender: TObject; Directory : TDirName);
begin
  EditDirectory.Text := Directory;
end;

end.

```

Voici aussi la forme texte du dfm :

## TestComposTutoMain.dfm

```

object FormMain: TFormMain
  Left = 0
  Top = 0
  BorderIcons = [biSystemMenu, biMinimize]
  BorderStyle = bsSingle
  Caption = 'Tests des composants du tutoriel'
  ClientHeight = 89
  ClientWidth = 425
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  Position = poScreenCenter
  OnCreate = FormCreate
  PixelsPerInch = 96
  TextHeight = 13
  object ButtonSelectDir: TButton
    Left = 16
    Top = 16
    Width = 225
    Height = 25
    Caption = 'S'#233'lectionner un r'#233'pertoire'
    TabOrder = 0
    OnClick = ButtonSelectDirClick
  end
  object EditDirectory: TEdit
    Left = 16
    Top = 48
    Width = 393
    Height = 21
    TabOrder = 1
    Text = 'EditDirectory'
  end
  object SelectDirDialog: TSelectDirDialog
    Caption = 'Vous pouvez choisir un dossier'
    OnAccept = SelectDirDialogAccept
    Left = 248
    Top = 16
  end
end
end

```

Remarquez au passage l'effet des spécificateurs **default** et **stored**.

## IV-G - Une icône pour le composant

À nouveau, il reste une imperfection : ne trouvez-vous pas dommage que l'icône du composant soit un peu basique ? Nous allons remédier à cela.

Il nous faut tout d'abord une image. Celle-ci doit se trouver dans une ressource de type **dcr** (pour **Delphi Component Resource**). Pour éditer ce type de fichier, vous aurez besoin de l'éditeur d'images de Borland.

Dans Delphi, sélectionnez le menu **Outils|Éditeur d'images**. L'éditeur d'images s'ouvre. Sélectionnez le menu **Fichier|Nouveau|Ressources composants (\*.dcr)**. Une nouvelle fenêtre s'affiche. Faites alors **Ressource|Nouvelle|Bitmap**. Sélectionnez une taille de 24\*24 pixels. Nommez la ressource du nom du composant correspondant.

*Si l'unité enregistre plusieurs composants, le fichier .dcr peut contenir plusieurs bitmaps, chacun du nom du composant correspondant.*

La transparence est déterminée d'après la couleur du pixel inférieur gauche. Dessinez une image correcte pour votre composant, puis enregistrez le tout sous le nom de **RegComposTuto.dcr** (vous devez utiliser le même nom de fichier que celui de l'unité qui **enregistre** les composants avec l'extension .dcr).

Finalement, rouvrez le paquet **ComposTutoD.bpl**, supprimez l'unité **RegComposTuto** puis rajoutez-là : le fichier .dcr est ajouté automatiquement.

Recompilez. Le composant **TSelectDirDialog** est maintenant doté d'une icône digne de ce nom.

## V - Conclusion

Nous voici arrivés au terme de notre découverte de la création de composants non-visuels et des quelques techniques indispensables.

J'espère que vous aurez apprécié ce tutoriel et qu'il vous aura été bénéfique.

Vous pouvez [télécharger tous les sources présentés dans ce tutoriel](#), tels qu'ils sont à ce stade.

Si ce lien ne fonctionne pas chez vous, utilisez [celui-ci](#).

## VI - Liens utiles

Voici quelques liens utiles en rapport avec ce dont avons parlé dans cette partie :

- [Cours et tutoriels sur les packages](#)
- [Cours et tutoriels en rapport avec les composants](#)
- [Section Composants de la FAQ Delphi](#)
- [Introduction to Component Building](#)





## VII - Remerciements

Je voudrais adresser un très grand MERCI à [Laurent Dardenne](#) pour sa grande aide dans la réalisation de ce tutoriel, autant aux niveaux documentation, fond et forme.

Merci aussi à [gege2061](#) et [Bestiol](#) et encore une fois à [Laurent Dardenne](#) pour leurs relectures sur la forme et l'orthographe.