

# FMX : Recherche dans un TListView

## Utilisation de l'évènement OnFilter

Par Serge Girard  

Date de publication : 14 mai 2020

CONFIRMÉ

Le composant **TListView** permet d'ajouter très facilement une boîte de recherche par l'intermédiaire de sa propriété **SearchVisible**. Sans autre ligne de code la recherche s'effectue sur tous les textes (même invisibles) de la liste. Toutefois cette recherche est établie selon le principe : « accepter l'affichage si un des textes contient le filtre sans tenir compte de la casse ». L'évènement **OnFilter** permet de modifier ce principe.

Mon objectif est de montrer comment utiliser cet évènement, d'en démontrer le mécanisme et de fournir des moyens supplémentaires pour des recherches encore plus ciblées.

I - Bases de l'utilisation de TListView.....	3
I-A - Les différentes apparences.....	3
I-B - Apparence personnelle (custom).....	4
I-C - L'apparence dynamique.....	4
I-D - Le remplissage avec les LiveBindings.....	5
I-E - La boîte de recherche.....	6
II - L'événement OnFilter.....	9
II-A - Cas d'utilisation simple, une seule zone texte.....	9
II-B - Cas d'utilisation avec 2 zones texte.....	10
II-B-1 - Mécanisme de l'évènement.....	10
II-B-2 - Ébauche de solution.....	11
II-C - Nombre d'objets de type texte d'un élément de liste.....	12
II-D - Rechercher sur une zone particulière.....	14
III - Rendre plus « générique » la méthode.....	15
III-A - Première ébauche.....	16
III-A-1 - Utilisation.....	18
III-B - Le coin de l' « expert ».....	20
III-B-1 - Utilisation.....	25
IV - Conclusion.....	25

## I - Bases de l'utilisation de TListView

On trouve beaucoup de vidéos et d'aides sur ce composant mais souvent sans s'attarder sur une utilisation plus poussée. Pour repousser ces manques, j'ai, moi aussi, beaucoup écrit à son propos dans d'autres tutoriels :

- **Mettre de la couleur dans un TListView ;**
- **Personnaliser un TListView : ajouter des pieds de groupes ;**
- TListView apparence dynamique et images.

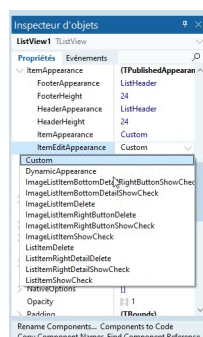
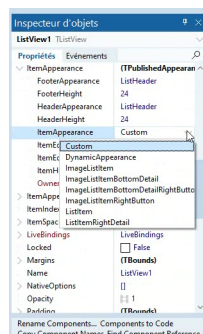
Et ai communiqué sur mon blog :

- **Obtenir une liste groupée avec les Livebindings ;**
- **Obtenir une liste groupée avec les Livebindings. Suite ;**
- **Obtenir une liste groupée avec les Livebindings. Finalisation ?**
- **ListView FMX, entêtes et pieds de groupes : Suite et Fin !**
- **Un problème de ListView avec application de style enfin résolu ;**
- **Modifier la hauteur (et accessoirement d'autres propriétés) de la boîte de recherche d'une liste.**

Les différentes sections qui suivent vont surtout mettre en exergue ce qui pourrait influencer sur la recherche au sein des éléments de la liste.

**i** *Ce chapitre n'est qu'un survol du TListView. Vous trouverez une documentation relativement complète des possibilités de ce composant dans le **docwiki d'Embarcadero***

### I-A - Les différentes apparences



La plupart des apparences proposées ne contiennent que l'élément texte :

<b>TAppearanceNames</b> avec une seule zone texte
ListItem, ListItemDelete, ListItemShowCheck
ImageListItemRightButton, ImageListItemRightButtonDelete, ImageListItemRightButtonShowCheck
ImageListItem, ImageListItemDelete, ImageListItemShowCheck

Ou offrent une zone détail en supplément :

<b>TAppearanceNames</b> avec une zone détail en supplément.
ImageListItemBottomDetail, ImageListItemBottomDetailShowCheck
ListItemRightDetail, ListItemRightDetailDelete, ListItemRightDetailShowCheck
ImageListItemBottomDetailRightButton, ImageListItemBottomDetailRightButtonShowCheck

## I-B - Apparence personnelle (custom)

Cette apparence prédéfinie est particulière, applicable également aux apparences d'entête (**HeaderAppearance**) et de pied (**FooterAppearance**) de groupe. Elle serait équivalente à l'apparence **ImageListItemBottomDetailRightButtonShowCheck** certains éléments étant non visibles par défaut.



*La recherche ne s'effectuera jamais sur les entêtes et pieds de groupes. Si, toutefois, vous vouliez faire une recherche sur le texte contenu dans ceux-ci la solution consistera à utiliser une apparence dynamique et ajouter un **TTextObjectAppearance***

## I-C - L'apparence dynamique

Apparue plus tardivement (Delphi Berlin 10.1), cette apparence permet un design beaucoup plus personnalisé et mettre, ainsi, plus de deux zones de texte.



*Pour plus d'informations sur l'apparence dynamique, le plus simple est de commencer par lire ce que contient le Docwiki Embarcadero à ce sujet [à cette adresse](#).*

Très succinctement, puisque ce n'est pas le sujet principal, après avoir sélectionné pour la propriété **ItemAppearance** la valeur **DynamicAppearance**, il est possible d'ajouter des objets spécifiques (ou plutôt des apparences d'objets) les uns à la suite des autres.

Cinq types d'objets sont possible :

<b>TTextObjectAppearance</b>	Un objet texte, le seul qui nous intéressera dans le cadre de ce tutoriel
<b>TImageObjectAppearance</b>	Un objet image
<b>TAccessoryObjectAppearance</b>	Un objet accessoire, trois types : <b>More</b> , <b>Checkmark</b> , <b>Detail</b>
<b>TTextButtonObjectAppearance</b>	Un objet bouton avec texte
<b>TGlyphButtonObjectAppearance</b>	Un objet bouton, trois types : <b>Add</b> , <b>Delete</b> , <b>CheckBox</b>



Certains de ces objets ne sont visibles que si la liste est en mode édition. Le terme de ce mode est trompeur, vous ne pourrez pas pour autant, modifier les textes comme il serait possible de le faire dans une grille.

Une fois les objets ajoutés, avec un peu d'adresse, il est possible d'arranger les tailles, positions et alignements de ceux-ci.

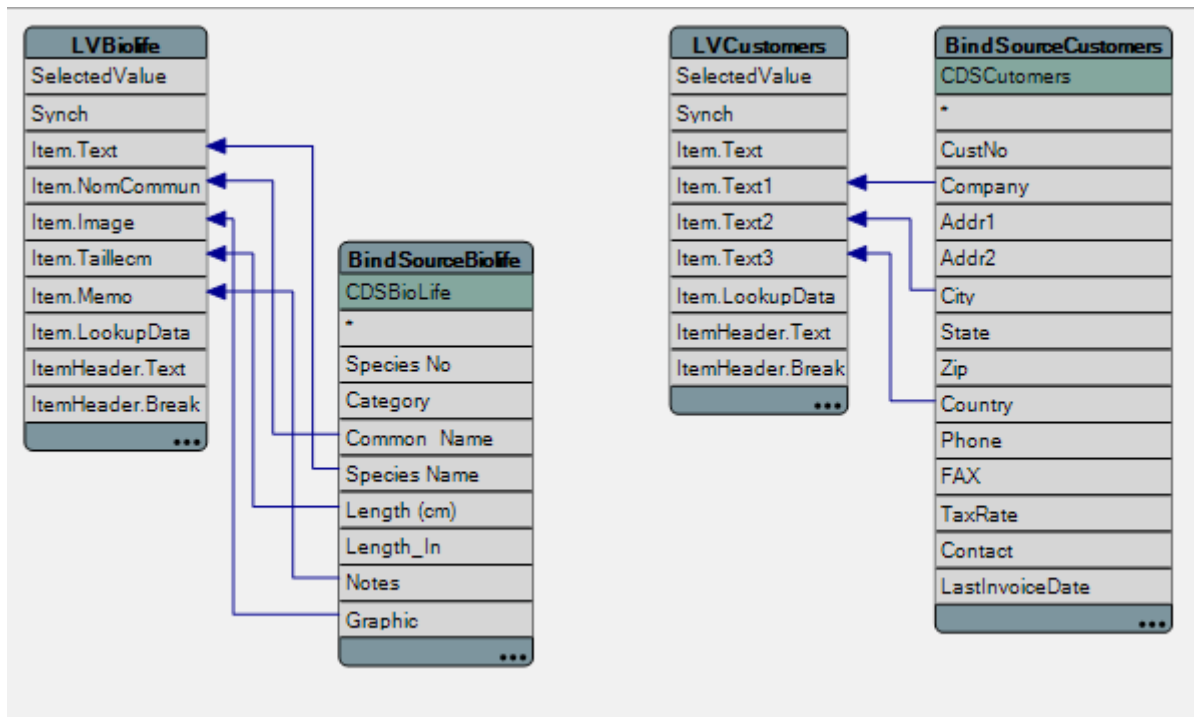
## I-D - Le remplissage avec les LiveBindings

Qui dit **TListView** sous-entend le plus souvent un grand nombre de données, contrairement à un **TListBox**. L'éditeur de liaison nous permet de l'alimenter facilement.

Dans les projets illustrant ce tutoriel, j'utiliserai essentiellement un composant **TClientDataSet** pour accéder aux données de deux fichiers situés dans le répertoire (**\$Samples**)**Data** : **Biolife.xml** et **Customers.xml**.



Sélectionnez la liste et utilisez le menu contextuel (option : lier visuellement) pour réaliser les liens entre liste et données. Ce n'est l'affaire que d'une ou deux minutes pour obtenir une liste liée aux données si votre table est ouverte.



Concepteur Livebindings projet RechercheBase



Cela ne veut pas dire que vous ne pourrez pas faire de recherche dans une liste remplie par code. Juste qu'il est très facile de remplir une liste sans code grâce aux LiveBindings.

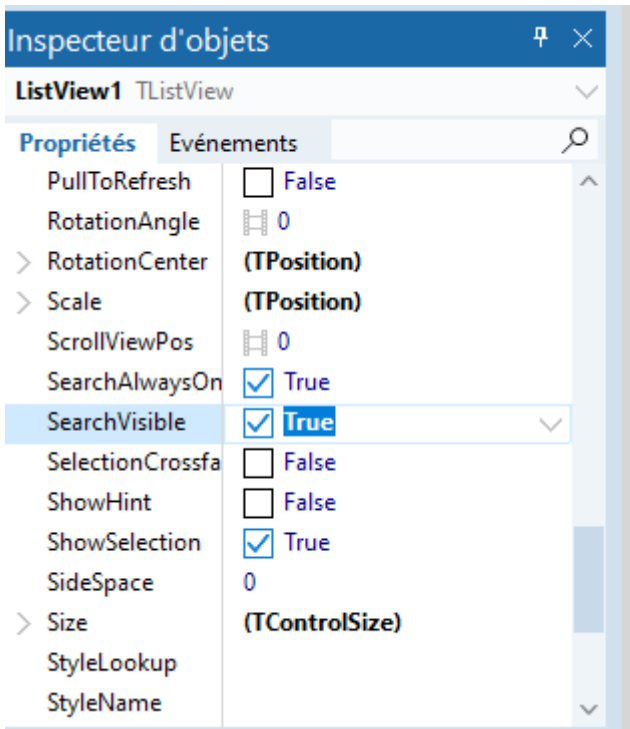
Vous retrouverez également de nombreux tutoriels plus orientés Livebindings sur [mon site](http://serge-girard.developpez.com/tutoriels/Delphi/FMX/Listes/RechercherDans/).

## I-E - La boîte de recherche



Pour illustrer ce chapitre, un projet nommé **RechercheBasique** : [téléchargeable ici](#)

Ajouter une boîte de recherche à un TListView est hyper facile puisqu'il suffit d'activer cette fonctionnalité en cochant la propriété **SearchVisible** de celle-ci.



Ajout boîte de recherche

Lorsqu'il n'y a qu'un seul élément la question de savoir sur quel texte sera faite une recherche ne se pose même pas. Par contre, lorsque ces éléments contiennent plus d'une zone contenant du texte, la recherche, par défaut, sera faite sur l'ensemble de ces objets texte (visibles ou non).

Pratique dans la plupart des cas cet effet peut se révéler gênant dans certains cas.

Deux illustrations à ce propos :

- Dans ma liste de clients je veux retrouver les clubs d'un pays (i.e. le Venezuela), taper seulement « **ven** » (vous noterez au passage que la recherche n'est pas sensible à la casse), me fournit six résultats alors qu'un seul était attendu. Bien sûr taper « **vene** » me fournira la bonne réponse.




Mais, si je cache la zone pays (je vous rappelle que la recherche se fait même sur les éléments cachés), vous avouerez que l'utilisateur sera mal renseigné.

Exemples de recherches		Exemples de recherches	
Customers	Biolife	Customers	Biolife
		Q ven	
ai Dive Shoppe	US	Adventure Undersea	B
aa Kauai		Belize City	
co	Bahamas	SCUBA Heaven	Baha
port		Nassau	
t Diver	Cyprus	Marina SCUBA Center	Venez
o Paphos		Caracas	
man Divers World Unlimited	British West Indies	Divers of Venice	
nd Cayman		Venice	
Sawyer Diving Centre	US Virgin Islands	Vashon Ventures	
istiansted		Honolulu	
Jack Aqua Center	US	Ocean Adventures	
oahu		Maui	
Divers Club	US Virgin Islands		
istiansted			

- Second exemple, j'affiche aussi la description de l'animal. Objectif avoué : obtenir tous les poissons dont le nom contient « **blu** ».

Exemples de recherches

Customers Biolife




Clown Triggerfish  
50

known as the big spotted triggerfish. Inhabits outer reef areas and s upon crustaceans and mollusks by crushing them with powerful n. They are voracious eaters, and divers report seeing the clown erfish devour beds of pearl oysters.

not eat this fish. According to an 1878 account, "the poisonous flesh primarily upon the nervous tissue of the stomach, occasioning violent ms of that organ, and shortly afterwards all the muscles of the body. frame becomes rocked with spasms, the tongue thickened, the eye d, the breathing laborious, and the patient expires in a paroxysm of me suffering."

edible.

ge is Indo-Pacific and East Africa to Somoa.




Red Emperor  
60

ed seaperch in Australia. Inhabits the areas around lagoon coral reefs sandy bottoms.

red emperor is a valuable food fish and considered a great sporting that fights with fury when hooked. The flesh of an old fish is just as

Exemples de recherches

Customers Biolife




Blue Angelfish  
30

Habitat is around boulders, caves, coral ledges and crevices in shallow waters. Swims alone or in groups.

Its color changes dramatically from juvenile to adult. The mature adult fish can startle divers by producing a powerful drumming or thumping sound intended to warn off predators.

Edibility is good.

Range is the entire Indo-Pacific region.



Cabezon  
99

Often called the great marbled sculpin. Found over rocky or shell-encrusted bottoms from shallow to moderately deep waters. It feeds primarily on crustaceans and mollusks.

The male cabezon will not budge while guarding its nest and can even be touched by divers.

Edibility is good; the flesh is bluish-green but turns white when cooked. The eggs of the cabezon are poisonous.

Range is from Alaska to Central Baja.

*Vous admettez que **Cabezon** ne rentre pas vraiment dans ces critères.*

**i** *Bien sûr, encore une fois, augmenter mes critères de recherches, rechercher « **blue** » fera disparaître **Cabezon** de la liste.*

En conclusion de cette petite démonstration ([téléchargeable ici](#)) ce qu'il nous faut, c'est un moyen de pouvoir affiner nos critères de recherche. Par exemple :

- Que celle-ci soit sensible à la casse ;
- Que celle-ci ne sélectionne l'élément que s'il commence par la valeur recherchée ;
- Que la sélection de soit faite que sur quelques éléments mais pas tous ;
- Et coëtera.

C'est là que va intervenir la codification dans l'évènement **OnFilter** de **TListView**.





Pour ne pas déroger au **principe de Pareto** (loi des 80-20) dans au moins 80 % des cas où vous aurez mis une possibilité de recherche dans une liste il n'y aura pas besoin de personnaliser cette recherche. La suite de ce tutoriel tente de répondre aux 20 % restants.

## II - L'événement OnFilter

*Note au lecteur :*



À partir de maintenant, une fois établi que la recherche ne se fait que dans le texte contenu dans les éléments de type **TTextObjectAppearance**, dans le chapitre qui suit, objet et objet texte seront des synonymes.

Intéressons-nous aux paramètres de cet événement

Nom	Type	
Sender	TObject	Cela va de soi, il s'agit de l'objet à qui appartient l'évènement, donc un TListView
<b>const</b> AFilter	String	Le texte à rechercher
<b>const</b> AValue	String	Le point délicat ! D'où vient cette valeur et quid en cas de multiples objets contenant du texte ?
<b>var</b> Accept	Boolean	Variable qui va indiquer si l'on accepte ou pas l'élément selon la condition que l'on indiquera dans le code.

### II-A - Cas d'utilisation simple, une seule zone texte

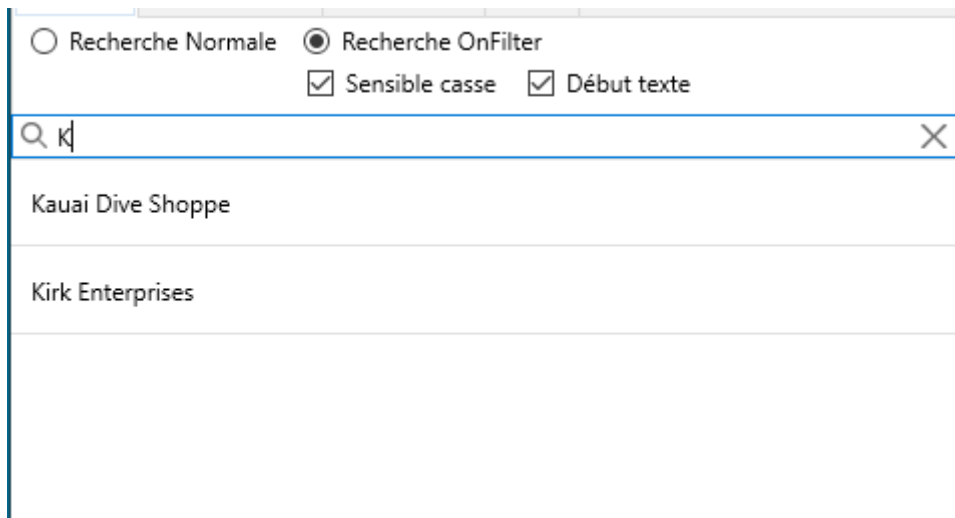
Pour illustrer l'utilisation de l'évènement, intéressons-nous à une recherche de début de texte.

```
procedure TForm1.ListView1Filter(Sender: TObject; const AFilter, AValue: string; var
  Accept: Boolean);
begin
  Accept:=AFilter.IsEmpty OR AValue.StartsWith(AFilter, false);
  // le second paramètre de StartsWith permet d'indiquer la sensibilité
  // à la casse
end;
```

*Simplissime non ?*



Vous retrouverez ce code dans le projet **RechercheOnFilter** [téléchargeable ici](#)



☐ Recherche Normale    ☒ Recherche OnFilter  
☒ Sensible casse    ☒ Début texte

Q K

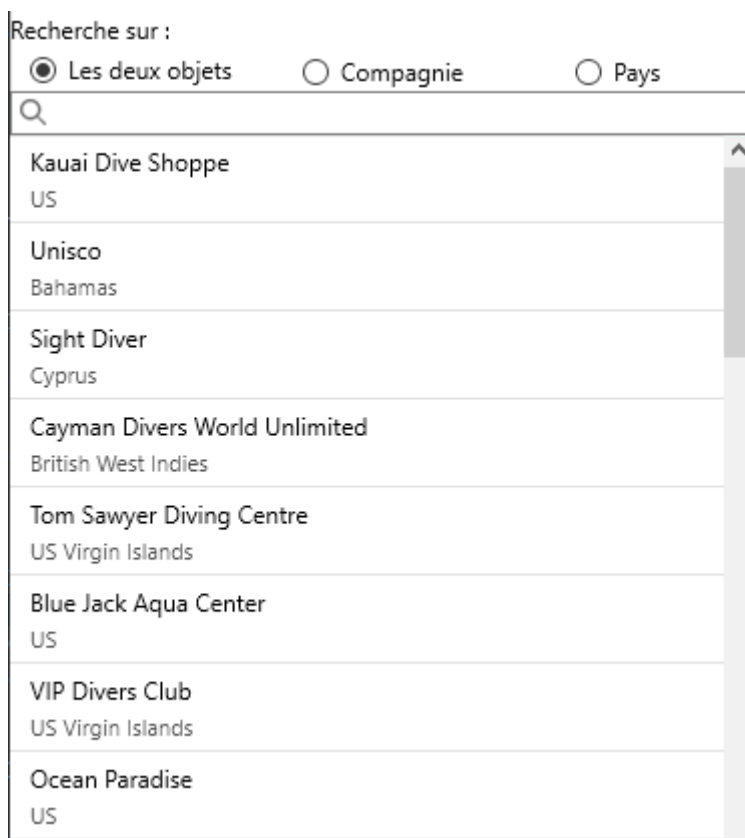
Kauai Dive Shoppe

Kirk Enterprises

*OnFilter un seul objet*

## II-B - Cas d'utilisation avec 2 zones texte

Prenons maintenant le cas d'un affichage de type **ImageListItemBottomDetail**



Recherche sur :

☒ Les deux objets    ☐ Compagnie    ☐ Pays

Q

Kauai Dive Shoppe  
US

Unisco  
Bahamas

Sight Diver  
Cyprus

Cayman Divers World Unlimited  
British West Indies

Tom Sawyer Diving Centre  
US Virgin Islands

Blue Jack Aqua Center  
US

VIP Divers Club  
US Virgin Islands

Ocean Paradise  
US

Le problème principal est de déterminer comment est fourni **AValue**.

### II-B-1 - Mécanisme de l'évènement

Pour étudier ce mécanisme, il suffit de poser un point d'arrêt au sein de la fonction et de contrôler le contenu de la constante.

```

- | procedure TMainForm.LV2ObjetsFilter(Sender: TObject; const AFilter,
70 | AValue: string; var Accept: Boolean);
- | begin
- |   if AFilter.IsEmpty then Exit;
73 | Accept:= AValue.StartsWith(AFilter)
- | end;

```

Première constatation, l'évènement est fréquemment levé, en fait à chaque lecture d'un objet texte. Ainsi, successivement, le contenu de la constante **AValue** sera-t-il la valeur de l'objet **text** puis de l'objet **detail**.



*Successivement ? Non, pas tout à fait, si le premier objet testé satisfait la condition alors le second objet n'est pas testé et le programme passe tout de suite à l'élément de liste suivant.*

## II-B-2 - Ébauche de solution

A priori, il n'y a aucun moyen de savoir quel objet est en cours de dessin (du moins de manière simple). Introduire une sorte de compteur d'objets testés fut donc ma première approche.

### Ajout compteur

```

var
  MainForm: TMainForm;
  ItemObjectNumber : word;

```

### Initialisation compteur

```

initialization
  ItemObjectNumber:=0;
finalization
end.

```

Reste à charge de coder l'incrémement et la réinitialisation de ce compteur au sein de l'évènement **OnFilter**.

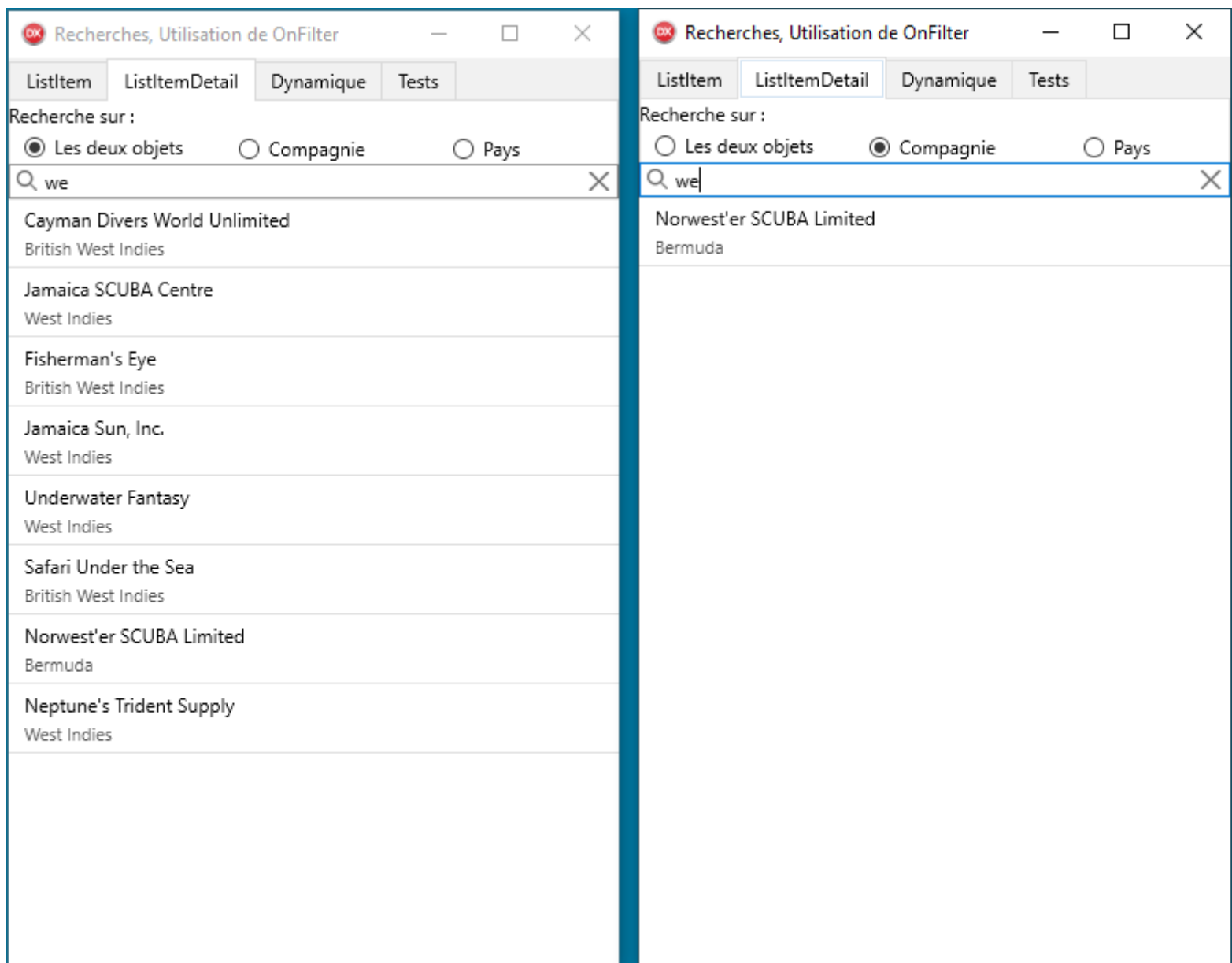
### OnFilter

```

procedure TMainForm.LV2ObjetsFilter(Sender: TObject; const AFilter,
  AValue: string; var Accept: Boolean);
begin
  if AFilter.IsEmpty then Exit; // pas de recherche
  inc(ItemObjectNumber); // incrémement
  // tests en fonction du mode
  if RBTexte.IsChecked then Accept := (ItemObjectNumber=1) AND
    UpperCase(AValue).Contains(UpperCase(AFilter));
  if RBDetail.IsChecked then Accept := (ItemObjectNumber=2) AND
    UpperCase(AValue).Contains(UpperCase(AFilter));
  // réinitialisation si le nombre d'objets texte est atteint
  // ou si la condition est réalisée (Accept:=true)
  if (ItemObjectNumber=2) OR Accept then ItemObjectNumber:=0;
end;

```

Dans ce contexte (toute apparence à base de **ListItemDetail** ou plus exactement ne contenant que deux objets de type texte) cette solution est fonctionnelle.



Elle peut même être applicable à de plus grands nombres d'objets de type texte, pour peu de bien maîtriser les diverses valeurs de la variable **ItemObjectNumber**.

En « puriste » je n'y vois que quelques bémols :

- 1 Le fait d'avoir à renseigner le nombre d'objets maximum **ItemObjectNumber=2**
- 2 Connaître l'ordre dans lequel seront affichés les objets de l'élément **Accept := (ItemObjectNumber=1) AND ...**
- 3 Le fait d'être obligé d'utiliser une variable globale (le compteur d'objets).

## II-C - Nombre d'objets de type texte d'un élément de liste

Le premier bémol peut, avec un peu d'expertise, être contourné en introduisant une variable privée (voire une propriété) calculée avant une quelconque opération de recherche sur la liste.

Pour les apparences dites « classiques », cf. **I.A**, le nombre d'objets est déductible facilement en recherchant dans le type d'apparence ( **<liste>.ItemAppearance.ItemAppearance** ) s'il contient le texte « **detail** » ou non.

```
nombredobjets:=1;
if LowerCase(aList.ItemAppearance.ItemAppearance).Contains('detail')
then Inc(nombredobjets);
```

Pour une apparence dynamique il va falloir vérifier au sein de la collection d'objets associée.

```
nombredobjets:=0;
if aList.ItemAppearance.ItemAppearance='DynamicAppearance'
then begin // apparence dynamique
  DynApp:=TDynamicAppearance(aList.ItemAppearanceObjects.ItemObjects);
  for appObj in DynApp.ObjectsCollection do
  begin
    if (appobj is TAppearanceObjectItem) AND
      (TAppearanceObjectItem(AppObj).Appearance is TTextObjectAppearance)
    then inc(nombredobjets);
  end;
end ;
```



*Pour pouvoir accéder à la collection d'objets il est impératif d'ajouter dans les clauses d'utilisation (uses) l'unité **FMX.ListView.DynamicAppearance***

Le tout peut-être intégré dans une fonction plus générale :

#### Nombre d'objets texte

```
uses FMX.ListView.DynamicAppearance;
...

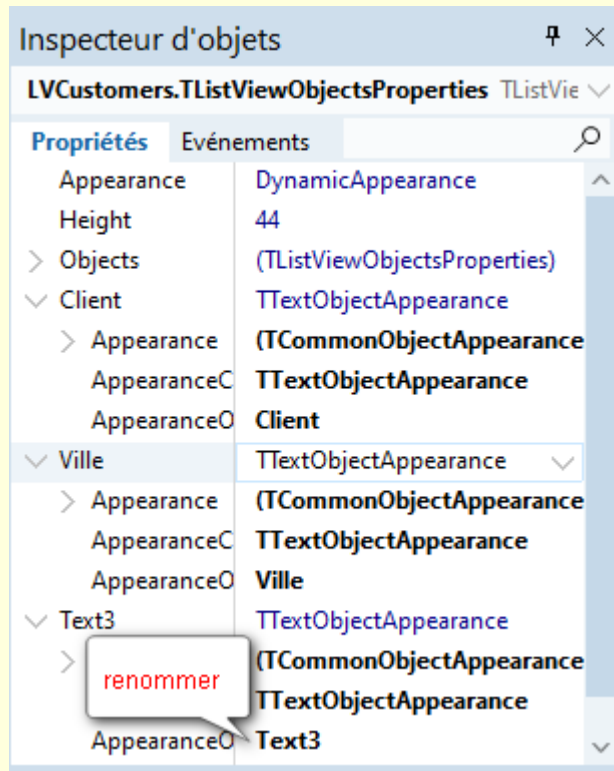
function TMainForm.HowManyTextObjects(const aList: TListView): integer;
var DynApp : TDynamicAppearance;
    appObj : TCollectionItem;
begin
  result:=0;
  if aList.ItemAppearance.ItemAppearance='DynamicAppearance'
  then begin
    DynApp:=TDynamicAppearance(aList.ItemAppearanceObjects.ItemObjects);
    for appObj in DynApp.ObjectsCollection do
    begin
      if (appobj is TAppearanceObjectItem) AND
        (TAppearanceObjectItem(AppObj).Appearance is TTextObjectAppearance)
      then inc(result);
    end;
  end
  else begin
    result:=1;
    if LowerCase(aList.ItemAppearance.ItemAppearance).Contains('detail')
    then Inc(result);
  end;
end;
```

ListItem	ListItemDetail	Dynamique	Tests
ListItem contient 1 objet			
ListItemDetail contient 2 objets			
Combien d'objets texte dans la liste Dynamique			

## II-D - Rechercher sur une zone particulière

Par zone, j'entends selon le nom de l'objet texte.

*D'où l'importance de nommer ces objets plutôt que de laisser les noms par défaut : **Text1...** **Textn**, mais surtout, faites-le avant d'établir les liaisons !*



*Je n'ai pas encore trouvé de solution pour utiliser les noms de colonnes de la table liée. Même si cela doit être possible, je gage que cela compliquerait énormément le code.*

Une fois l'astuce d'obtention des objets d'une liste d'apparence dynamique connue il est simple de retrouver les noms des objets de la collection, la propriété **AppearanceObjectName**.

```
procedure GetTextObjectNames(aList : TListView);
var DynApp : TDynamicAppearance;
    appObj : TCollectionItem;
begin
    if aList.ItemAppearance.ItemAppearance='DynamicAppearance'
    then begin
        DynApp:=TDynamicAppearance(AList.ItemAppearanceObjects.ItemObjects);
        for appObj in DynApp.ObjectsCollection do
            begin
                if appObj is TAppearanceObjectItem then
                    if TAppearanceObjectItem(AppObj).Appearance is TTextObjectAppearance
                    then begin
                        Memo1.Lines.Add(TAppearanceObjectItem(AppObj).AppearanceObjectName);
                    end;
                end;
            end
        else begin
            Memo1.Lines.Add('text');
            if LowerCase(aList.ItemAppearance.ItemAppearance).Contains('detail')
```

```
then Memo1.Lines.Add('detail');
end;
end;
```

La liste Dynamique contient 3 objets

Noms des objets texte

☐ ListItem
 ☐ ListItemDetail
 ☒ Liste Dynamique

Client  
Ville  
Pays

Pour peu de stocker ces noms des zones qui doivent être filtrées dans une liste, j'ai ainsi la réponse à mon deuxième bémol du chapitre [II.B.2](#) en remplaçant

Accept := (ItemObjectNumber=1) AND ... par une instruction qui recherchera si l'objet en cours fait partie de cette liste ListeDesZonesATester.Contains(nomdelazoneencours) ;

*Encore un peu nébuleux ? Tout va se dévoiler dans le prochain chapitre.*



*Passez quand même un peu de temps sur ce second projet exemple, [téléchargeable ici](#), pour voir le comportement des recherches.*

### III - Rendre plus « générique » la méthode

Pourquoi rendre plus générique la méthode ? Tout simplement parce qu'ainsi stockées dans une unité à part elle pourra être utilisée dans d'autres formes d'une application ou, même, d'autres applications.

Reprenons les besoins, il nous faut :

- 1 Pouvoir spécifier un mode de recherche. Par défaut le texte de recherche doit être contenu (**Contains**). Dans le chapitre [II.A](#) j'ai fait la recherche sur le début de valeur (**StartsWith**). En plus de cette possibilité d'autres fonctions sont envisageables comme l'égalité (=) ou la recherche en fin de valeur (**EndsWith**).
- 2 Il faut également pouvoir indiquer si la recherche sera sensible ou non à la casse.
- 3 Intégrer le compteur d'objets tel que j'ai pu le définir au chapitre [II.B](#)
- 4 Faire en sorte d'obtenir le nombre d'objets de type texte cf. chapitre [II.C](#)
- 5 Enfin, la possibilité de rechercher sur une ou plusieurs zones de type texte doit faire partie de l'arsenal. Pour cela j'aurais besoin de deux listes, une contenant tous les noms d'objets de type texte d'un élément de liste, l'autre pour contenir le nom des différents objets que l'on veut prendre en compte (chapitre [II.D](#)).

### III-A - Première ébauche

Ma première approche a été de créer une nouvelle classe que je nommerai **TSearchInList** pour surclasser la recherche originelle.

Pour répondre au point numéro 1, je vais utiliser une énumération **TSearchMode**.

#### Énumération des modes de recherche.

```
TSearchMode = (smContains, smStartwith, smEndwith, smEqual);
```

Ma nouvelle classe sera composée de deux parties :

- Une partie privée.

Membres	Types	
ParentList	TListView	Le parent de la classe, la liste appelante.
Fields	TList<String>	Liste des objets texte
CurIndice	SmallInt	Index de l'objet texte en cours

- Une partie publique.

Membres	Types	
Mode	TSearchMode	Le mode de recherche
CaseSensitive	Boolean	La sensibilité à la casse
TestFields (1)	TList<String>	La liste des noms des objets texte à tester
Constructor	-	Le constructeur de la classe
Destructor	-	Le destructeur de cette classe
Accept	function	Fonction à deux arguments <b>aValue</b> , <b>aFilter</b> renvoyant le résultat du test.

#### Déclaration de la classe.

```
TSearchInList = Class(TObject)
strict private
  ParentList : TListView;
  Fields : TList<String>;
  CurIndice : smallint;
public
  TestFields : TList<String>;
  Mode : TSearchMode;
  CaseSensitive : Boolean;
  constructor Create(AOwner : TListView);
  destructor Destroy; override;
  function Accept(const aFilter, aValue: string) : Boolean;
end;
```



*L'unité de définitions des apparences dynamiques est nécessaire.*



**uses** FMX.ListView.DynamicAppearance;

**constructor** Create(AOwner : TListView);

Le constructeur va se charger d'initialiser nombre d'éléments dont :

- Le mode de recherche par défaut (**smContains**) ;
- La création des différentes listes **Fields** et **TestFields** ;
- Le chargement de la liste (**Fields**) contenant les noms des objets texte trouvés ;
- Et, en dernier lieu, la variable **CurIndex** sera mise à zéro.

#### Constructeur.

```

constructor TSearchinList.Create(AOwner: TListView);
var DynApp : TDynamicAppearance;
    appObj : TCollectionItem;
begin
    inherited Create;
    Parent:=AOwner;
    CurIndex:=0;
    Mode:=smContains;
    TextObjectsNames:=TList<String>.Create;
    // TestFields sera une liste insensible à la casse
    Objects2Test:=TList<String>.Create (TComparer<String>.Construct (
        function(const s1, s2: String): Integer
        begin
            Result := CompareText(s1, s2) ;
        end));
    // Remplissage de la liste des noms des TTextObjectAppearance
    if Parent.ItemAppearance.ItemAppearance='DynamicAppearance'
    then begin
        DynApp:=TDynamicAppearance (Parent.ItemAppearanceObjects.ItemObjects);
        for appObj in DynApp.ObjectsCollection do
            begin
                if appObj is TAppearanceObjectItem then
                    if TAppearanceObjectItem (AppObj) .Appearance is TTextObjectAppearance
                    then TextObjectsNames.Add (TAppearanceObjectItem (AppObj) .AppearanceObjectName);
                end;
            end begin
                TextObjectsNames.Add ('text');
                if LowerCase (Parent.ItemAppearance.ItemAppearance) .Contains ('detail')
                then begin
                    TextObjectsNames.Add ('detail');
                end;
            end;
        CaseSensitive:=False;
    end;

```



*Notez le petit truc supplémentaire, au niveau de la liste des noms des objets à tester (**FObjects2Test**), l'ajout d'un comparateur permettant l'insensibilité à la casse des éléments.*

```

Objects2Test:=TList<String>.Create (TComparer<String>.Construct (
    function(const s1, s2: String): Integer
    begin
        Result := CompareText(s1, s2) ;
    end));

```

**destructor** Destroy; **override**;

Le destructeur aura pour tâche de détruire les listes créées et, par précaution (2) , tout autre objet qui aurait pu être créé auparavant.

### Constructeur.

```
destructor TSearchinList.Destroy;
begin
  TextObjectsNames.Free;
  Objects2Test.Free;
  inherited Destroy;
end;
```

**function** Accept(const aFilter, aValue: string) : Boolean;

La fonction **Accept** est, en quelque sorte, le cœur de la classe puisque cette fonction va surcharger la fonction de base utilisée par le **TSearchBox** « conventionnel ».

### Fonction Accept.

```
function TSearchinList.Accept(const AFilter, AValue: string): Boolean;
var AVal, AFil, test : String;
begin
  if AFilter.IsEmpty then Exit(True);
  Result:=False;
  // si aucun objet texte spécifique demandé alors tous les objets
  if (Objects2Test.Count=0)
    OR (Objects2Test.IndexOf(TextObjectsNames[CurIndice])>-1)
  then begin
    AVal:=AValue;
    AFil:=AFilter;
    // sensibilité à la casse
    if not casesensitive then
      begin
        AVal:=LowerCase(AValue);
        AFil:=LowerCase(AFilter);
      end;
    // mode de recherche
    case Self.Mode of
      smContains : Result:= AVal.Contains(AFil);
      smStartwith : Result:= AVal.StartsWith(AFil);
      smEndwith : Result:= AVal.EndsWith(AFil);
      smEqual : Result:= AVal=AFil;
    end;
  end;
  if Result then CurIndice:=0 // trouvé
    else inc(CurIndice); // prochain objet
  if CurIndice>TextObjectsNames.Count-1 then CurIndice:=0; // tous les objets texte ont été testé
end;
```

## III-A-1 - Utilisation

- 1 Ajouter l'unité **SearchInListView** à la liste des unités utilisées (partie interface).

```
interface

uses
  ...
  SearchInListView;
```

- 2 Déclarer une variable , privée ou publique, de type **TSearchInList**.

```
private
  { Déclarations privées }
  Search : TSearchInList;
```

- 3 Créer la variable lors de l'évènement **OnCreate** de la forme.

### onCreate

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Search:=TSearchInList.Create(ListView1);
```

#### onCreate

```
// Case Sensitive False; par défaut
end;
```

#### 4 Codifier l'évènement **OnFilter** de la liste.

#### onFilter

```
procedure TForm1.ListViewFilter(Sender: TObject; const AFilter,
  AValue: string; var Accept: Boolean);
begin
  Accept:=Search.Accept(AFilter,AValue);
end;
```

C'est prêt, il ne reste plus qu'à fournir à l'objet les propriétés en fonction de la demande

Exemples :

- Changer la sensibilité à la casse

```
Search.CaseSensitive:=chkCasse.IsChecked;
```

- Changer les zones de recherches

```
Search.TestFields.Clear;
Search.TestFields.Add('company');
Search.TestFields.Add('country');
```

- Changer le mode de recherche

```
Search.Mode:=TSearchMode.smStartwith;
```

En fin de programme, ne pas oublier de détruire la variable de type **TSearchInList** créée pour éviter toute fuite de mémoire.

#### Clôture

```
procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Search.Free;
end;
```



***[Vous retrouverez l'application de cette unité dans le projet \*\*ListViewGenericSearch1\*\* partie d'un ensemble de sources téléchargeables \[ici\]\(#\)](#)***

**Recherche - Générique I**

Recherche sur :

☐ Tous
 ☐ Compagnie
 ☐ Ville
 ☒ Pays

☒ Sensible à la casse

☐ Contenant
 ☐ Débutant par
 ☒ Egal à
 ☐ se Terminant par

Kauai Dive Shoppe <i>Kapaa Kauai</i>	US
Blue Jack Aqua Center <i>Waipahu</i>	US
Ocean Paradise <i>Kailua-Kona</i>	US
The Depth Charge <i>Marathon</i>	US
Blue Sports <i>Giribaldi</i>	US
Makai SCUBA Club <i>Kailua-Kona</i>	US
Action Club <i>Sarasota</i>	US
Island Finders <i>St Simons Isle</i>	US
Blue Sports Club <i>Largo</i>	US



Ce programme ne fonctionnera pas pour les mobiles (**FormClose** non pris en compte) ni pour toute version prenant en charge la gestion de mémoire ARC (**free** devrait être remplacé par **disposeof**).

### III-B - Le coin de l' « expert »

Le seul inconvénient, encore que minime, à l'ébauche présentée c'est qu'il faut déclarer une variable de type **TSearchInList** pour chacune des listes d'une unité pour peu, bien sûr, que nous ayons plusieurs **TListView** sur une même forme et que nous voulions implémenter cette fonctionnalité pour chacun de ces composants. Sans oublier, par la suite, de supprimer les variables créées.

C'est en reprenant mes divers exemples que j'ai pu constater ce problème. C'est à ce stade que mes autres recherches sur ce composant m'ont fait envisager une solution à base d'interface.

Pour ce faire, je suis reparti de la classe **TSearchInList** présentée chapitre **III.A**.

J'en ai extrait la partie interface au sens strict du terme pour en faire une première unité.

## SearchListViewInterface

```

unit SearchListViewInterface;

interface

uses System.SysUtils, System.Types, System.Classes,
    System.Generics.Collections;

type
    TSearchMode = (smContains, smStartwith, smEndwith, smEqual);

    ISearchInListView = interface
        ['{0AF62378-366E-4AED-9918-28F8BA5859A8}']

        function Accept(const AFilter, AValue: string): Boolean;
        function GetTextObjectsNames : TList<String>;
        function GetTestFields: TList<String>;
        procedure SetTestFields(Value: TList<String>);
        function GetMode: TSearchMode;
        procedure SetMode(Value: TSearchMode);
        function GetCaseSensitive: Boolean;
        procedure SetCaseSensitive(Value: Boolean);
        property TextObjectsNames : TList<String> read GetTextObjectsNames;           property
        TestObjectsTextName : TList<String> read GetTestFields write SetTestFields;
        property Mode: TSearchMode read GetMode write SetMode;
        property CaseSensitive: Boolean read GetCaseSensitive write SetCaseSensitive;
    end;

implementation

end.
```

Puis j'ai créé une unité, une sorte d'adaptateur, qui va être utilisée pour surclasser le composant **TListView**.



*Ne vous étonnez pas des similitudes de code au niveau des procédures et fonctions, pour partie déjà ébauchées chapitre [III.A](#)*

Cette unité me servira à adjoindre l'interface au composant **TListView**.

**TListView** = **Class**(FMX.ListView.TListView, ISearchInListView)

Et ajouter les propriétés nécessaires.

## interface

```

unit ListViewSearchAdapter;

interface

uses System.SysUtils, System.Types, System.Classes,
    System.Generics.Collections, System.Generics.Defaults,
    FMX.ListView, SearchListViewInterface;

Type
    // surclassement de TListView, ajout de l'interface
    TListView = Class(FMX.ListView.TListView, ISearchInListView)
    strict private
        FCurrent: SmallInt;
        FObjectsNames : TList<String>;
    private
        FMode: TSearchMode;
        FCaseSensitive: Boolean;
        FObjects2Test : TList<String>;
    public
        // ajout des éléments de l'interface
```

## interface

```

constructor Create(AOwner: TComponent); override;
destructor Destroy; override;
function Accept(const AFilter, AValue: string): Boolean;
function GetTextObjectsNames : TList<String>;
function GetTestFields: TList<String>;
procedure SetTestFields(Value: TList<String>);
function GetMode: TSearchMode;
procedure SetMode(Value: TSearchMode);
function GetCaseSensitive: Boolean;
procedure SetCaseSensitive(Value: Boolean);
property Objects2Test : TList<String> read GetTestFields write SetTestFields;
procedure InitObjectsName;

end;
```

*C'est à dessein que le nom du composant restera identique à l'original.*

*Cependant, notez que la déclaration de la classe se fait en indiquant spécifiquement l'unité contenant **TListView***



**TListView** = **Class**(FMX.ListView.TListView, ISearchInListView)

*et non, comme vous pourriez en avoir l'habitude quand l'on veut « hacker » un composant pour accéder à des propriétés privées*

**THackListView** = **class**(TListView)

En préalable il va falloir ajouter quelques unités dans la liste des unités à utiliser

## implementation

```

implementation

uses FMX.ListView.Types,
      FMX.ListView.Appearances,
      FMX.ListView.DynamicAppearance;
```

**constructor** Create(AOwner: TComponent); **override**;

Toujours le même objectif, initialiser les propriétés c'est-à-dire les deux listes **FObjectsNames** et **FObjects2Test** équivalentes des deux listes **Fields** et **TestFields** de mon ébauche **III.A**, ainsi que les propriétés **Current**, **CaseSensitive** et **Mode**

```

constructor TListView.Create(AOwner: TComponent);
begin
  inherited;
  FObjectsNames:=TList<String>.Create;
  FObjects2Test:=TList<String>.Create(TComparer<String>.Construct (
    function(const s1, s2: String): Integer
    begin
      Result := CompareText(s1, s2) ;
    end));
  FCurrent:=0;
  FMode:=smContains;
  FCaseSensitive:=False;
end;
```

**destructor** Destroy; **override**;

Rien de nouveau, il faut supprimer la mémoire allouée par les objets (les listes) créées.

#### destructeur

```
destructor TListView.Destroy;
begin
    FreeAndNil(FObjectsNames);
    FreeAndNil(FObjects2Test);
    inherited;
end;
```

#### Les propriétés

Toute propriété, définie dans l'unité de l'interface nécessite un lecteur et, si besoin, un scribe (**getter**, **setter**).

```
// CaseSensitive
function TListView.GetCaseSensitive: Boolean;
begin
    Result:=FCaseSensitive;
end;

procedure TListView.SetCaseSensitive(Value: Boolean);
begin
    FCaseSensitive:=Value;
end;

// Mode
function TListView.GetMode: TSearchMode;
begin
    Result:=FMode;
end;

procedure TListView.SetMode(Value: TSearchMode);
begin
    FMode := Value;
end;

procedure TListView.SetTestFields(Value: TList<String>);
begin
    if not Assigned(FObjects2Test) then
        FObjects2Test := TList<string>.Create;
    FObjects2Test := Value;
end;

function TListView.GetTestFields: TList<String>;
begin
    result := FObjects2Test;
end;

function TListView.GetTextObjectsNames: TList<String>;
begin
    result:= FObjectsNames;
end;
```

#### procedure InitObjectsName;

C'est la petite nouvelle par rapport à l'ébauche. Son but, recenser les noms des objets texte.

Dans l'ébauche ce code se trouvait dans le constructeur.

#### InitObjectsNames

```
procedure TListView.InitObjectsName;
var
    DynApp: TDynamicAppearance;
    appObj: TCollectionItem;
begin
    if not Assigned(FObjectsNames) then
```

### InitObjectsNames

```

FObjectsNames := TList<String>.Create;
// obtenir les éléments texte
if ItemAppearance.ItemAppearance = 'DynamicAppearance' then
begin
  if EditMode
  then DynApp := TDynamicAppearance(ItemAppearanceObjects.ItemEditObjects)
  else DynApp := TDynamicAppearance(ItemAppearanceObjects.ItemObjects);
  for appObj in DynApp.ObjectsCollection do
  begin
    if appObj is TAppearanceObjectItem then
      if TAppearanceObjectItem(appObj).Appearance is TTextObjectAppearance
      then
        FObjectsNames.Add(TAppearanceObjectItem(appObj).AppearanceObjectName);
    end;
  end;
else
begin
  FObjectsNames.Add('text');
  if LowerCase(ItemAppearance.ItemAppearance).Contains('detail')
  then FObjectsNames.Add('detail');
end;
end;
end;

```

Elle ne sera appelée qu'une seule fois, à la première utilisation de la fonction **Accept**.

**function** Accept(const AFilter, AValue: string): Boolean;

Cette fonction ressemble beaucoup à celle de l'ébauche si ce n'est l'initialisation de la liste des noms des objets texte si celle-ci n'a pas été encore remplie et le nom d'une propriété qui a changée (**CurIndice** est devenue **FCurrent**).

### implémentation

```

function TListView.Accept(const AFilter, AValue: string): Boolean;
var
  Aval, AFil : String;
begin
  // récupère le nom des objets TTextObjectAppearance dans la liste
  if FObjectsNames.Count=0 then InitObjectsName;
  if AFilter.IsEmpty then Exit(True);
  Result:=False;
  // si aucun objet texte spécifique demandé alors tous les objets
  if (Objects2Test.Count=0)
  OR (Objects2Test.IndexOf(FObjectsNames[FCurrent])>-1)
  then begin
    Aval := AValue;
    AFil := AFilter;
    if not FCaseSensitive then
    begin
      Aval := LowerCase(AValue);
      AFil := LowerCase(AFilter);
    end;
    case FMode of
      smContains : Result := Aval.Contains(AFil);
      smStartwith: Result := Aval.StartsWith(AFil);
      smEndwith   : Result := Aval.EndsWith(AFil);
      smEqual     : Result := Aval = AFil;
    end;
  end;
  if Result then
    FCurrent := 0
  else begin
    inc(FCurrent);
    if FCurrent = (FObjectsNames.Count) then FCurrent := 0;
  end;
end;
end;

```





Les sources des deux unités se retrouvent dans le dossier compressé de l'application (**ListViewGenericSearch2**) [téléchargeable ici](#)

## III-B-1 - Utilisation

### 1 Déclarer l'utilisation des unités

```
interface
uses
...
FMX.ListView, SearchListViewInterface, ListViewSearchAdapter;
```



*Il est important de déclarer ces unités dans la partie interface, mais surtout, **après** l'utilisation de l'unité **FMX.ListView** pour que le surclassement opère.*

### 1 Dans le code, l'obligation restante est de codifier l'événement **OnFilter** de la liste.

```
// Codification de l'évènement OnFilter de la liste
procedure TForm1.ListView1Filter(Sender: TObject; const AFilter,
  AValue: string; var Accept: Boolean);
begin
  // appel de la fonction Accept de l'interface
  Accept:=ListView1.Accept(AFilter,AValue);
end;
```

Ensuite, selon les choix utilisateurs, on interviendra sur les propriétés. Ci-dessous quelques exemples :

#### exemples

```
// Changement de sensibilité à la casse (exemples)
ListView1.SetCaseSensitive(True);
ListView1.SetCaseSensitive(False);

// Changement de mode (exemples)
ListView1.SetMode(TSearchMode.smContains);
ListView1.SetMode(TSearchMode.smStartwith);
ListView1.SetMode(TSearchMode.smEqual);
ListView1.SetMode(TSearchMode.smEndwith);

// Objets à tester (exemples)
ListView1.Objects2Test.Add('text');
ListView1.Objects2Test.Add('detail');
ListView1.Objects2Test.Add('country');

// plusieurs objets (exemple)
ListView1.Objects2Test.AddRange(['city','country']);
```



Tout ceci est rassemblé dans une petite application (**ListViewGenericSearch2**) que vous pourrez [retrouver ici](#)

## IV - Conclusion

L'objectif de ce tutoriel était de démontrer les mécanismes de la recherche de TListView ainsi que les moyens de customiser celle-ci. Les différentes étapes permettant de comprendre la mise en œuvre et d'en apprendre un peu plus sur le surclassement et les interfaces.

Je le répète, dans la plupart des cas vous n'aurez pas besoin de surclasser ainsi votre TListView.

### Perspectives :

- Il serait sympathique de pouvoir mettre en exergue le texte recherché au sein des éléments, en utilisant les propriétés de **TTextLayout** ce devrait être envisageable (voir **chapitre VII.C de ce tutoriel**). Cependant un gros écueil se trouve sur la route, la recherche ne redessine pas l'élément de la liste mais, plutôt, l'occulte. Si, toutefois, je réussissais à le faire plus tard je ne manquerai pas de le communiquer sur mon blog.
- Bien que différent de **TListView**, il doit être possible d'utiliser la même technique pour un **TListBox**, peut-être le sujet d'un nouveau tutoriel.

Je remercie tous les intervenants qui ont pu m'aider à peaufiner chaque partie lors de mes appels à l'aide sur **le forum**. Ayez, comme moi, une pensée chaleureuse pour l'équipe rédactionnelle en particulier à **Malick**, les correcteurs techniques **gaby277**, **tourlourou** comme les correcteurs grammaticaux sans qui ce tutoriel ne pourrait être sous vos yeux.

- 1 : Une liste en prévision du besoin d'une recherche sur plus d'un objet texte.
- 2 : Je vous rappelle qu'il s'agit d'une ébauche, après réflexion l'**override** me semble inutile.