

Les génériques avec Delphi 2009 Win32

Avec en bonus les routines anonymes et les références de routine

par Sébastien Doeraene (sjrd.developpez.com)

Date de publication : 13 novembre 2008

Dernière mise à jour :

On les attendait depuis longtemps ! Les voici enfin : les génériques dans Delphi Win32. Ces petites merveilles arrivent avec Delphi 2009. Ce tutoriel vous propose de les comprendre, d'apprendre à les utiliser, puis à concevoir vos propres classes génériques.

 **Vos commentaires, critiques, suggestions, etc. sont les bienvenus sur le blog.**

Version anglophone de ce tutoriel - English version of this tutorial: **Generics with Delphi 2009 Win32**

Juan Badell a traduit ce tutoriel en espagnol : **Los genéricos en Delphi 2009 (miroir HTTP)**

Aleksey Timohin en a également fait **une traduction en russe.**

Cette traduction peut également être consultée au **format PDF**, construit par **Andrew Tishkin (miroir HTTP)**.

I - Introduction.....	3
I-A - Pré-requis.....	3
I-B - Que sont les génériques ?.....	3
II - L'utilisation au quotidien : l'exemple TList<T>.....	5
II-A - Un code simple de base.....	5
II-B - Assignations entre classes génériques.....	6
II-C - Les méthodes de TList<T>.....	7
II-D - TList<T> et les comparateurs.....	7
II-D-1 - Écrire un comparateur en dérivant TComparer<T>.....	8
II-D-2 - Écrire un comparateur avec une simple fonction de comparaison.....	9
III - Conception d'une classe générique.....	13
III-A - Déclaration de la classe.....	13
III-B - Implémentation d'une méthode.....	15
III-C - La pseudo-routine Default.....	16
III-D - Les références de routine avec les parcours d'arbre.....	16
III-D-1 - D'autres méthodes qui utilisent les routines anonymes ?.....	17
III-E - Et le reste.....	18
III-F - Comment utiliser TTreeNode<T> ?.....	18
IV - Conception d'un record générique.....	19
V - Contraintes sur les types génériques.....	21
V-A - Quelles sont les contraintes possibles ?.....	21
V-B - Mais à quoi ça sert ?.....	22
V-C - Une variante avec constructor.....	23
VI - Paramétrer une classe avec plusieurs types.....	24
VII - Autres types génériques.....	25
VIII - Méthodes génériques.....	26
VIII-A - Une fonction Min générique.....	26
VIII-B - La surcharge et les contraintes.....	27
VIII-C - Des ajouts pour TList<T>.....	28
IX - RTTI et génériques.....	30
IX-A - Les changements sur la pseudo-routine TypeInfo.....	30
IX-A-1 - Une fonction TypeInfo plus générale.....	31
IX-B - Les types génériques ont-ils des RTTI ?.....	31
X - Conclusion.....	33
XI - Télécharger les sources.....	34
XII - Remerciements.....	35

I - Introduction

I-A - Pré-requis

Ce tutoriel n'est pas destiné au débutant Delphi ou à celui qui n'a jamais pratiqué la programmation orientée objet. Il requiert de bien connaître le langage Delphi auparavant, et une connaissance non négligeable de la POO.

Concernant les génériques, aucune connaissance n'est *nécessaire* à la bonne compréhension et appréhension de ce tutoriel. Mais, au vu du nombre d'articles existant sur les génériques dans d'autres langages, je ne couvrirai pas en détails des questions du type "Comment bien utiliser les génériques ?". La prochaine section introduira rapidement le lecteur à la notion de générique, mais si vous ne savez pas encore ce que c'est, la meilleure compréhension viendra de la suite du tutoriel, par l'exemple.


Voici quelques articles traitant des génériques dans d'autres langages/plate-forme. Les concepts sont toujours les mêmes, et une grande partie de la mise en oeuvre également.

-  [Les génériques sous Delphi .NET](#) par Laurent Dardenne ;
-  [Generics avec Java Tiger 1.5.0](#) par Lionel Roux ;
-  [Cours de C/C++ - Les template](#) par Christian Casteyde.

I-B - Que sont les génériques ?

Les génériques sont parfois appelés *paramètres génériques*, un nom qui permet déjà bien mieux de les introduire. Contrairement au paramètre d'une fonction (argument), qui a une *valeur*, le paramètre générique est un *type*. Et un paramètre générique paramétrise une classe, une interface, un **record**, ou, moins souvent, une méthode.

Pour donner immédiatement une idée plus claire, voici un extrait de la classe **TList<T>**, que vous pourrez trouver dans l'unité **Generics.Collections.pas**.

 *Et oui, il est possible d'avoir des points dans le nom d'une unité (hors celui du .pas, j'entends). Cela n'a rien à voir avec la notion d'espace de noms de .NET ni celle de package de Java. C'est juste que le . a même valeur qu'un _ à ce niveau : il fait partie du nom, tout simplement.*

```

type
  TList<T> = class(TEnumerable<T>)
    // ...
  public
    // ...

    function Add(const Value: T): Integer;
    procedure Insert(Index: Integer; const Value: T);

    function Remove(const Value: T): Integer;
    procedure Delete(Index: Integer);
    procedure DeleteRange(AIndex, ACount: Integer);
    function Extract(const Value: T): T;

    procedure Clear;

    property Count: Integer read FCount write SetCount;
    property Items[Index: Integer]: T read GetItem write SetItem; default;
  end;
    
```

Vous pouvez directement observer l'étrangeté du type *T*. Mais est-ce vraiment un type ? Non, c'est un *paramètre générique*. Avec cette classe, si j'ai besoin d'une liste d'entiers, j'utilise **TList<Integer>**, au lieu d'une "simple" **TList** avec de nombreux transtypes nécessaires dans le code !

Les génériques permettent donc, en quelques sortes, de déclarer une *série de classes* (potentielles) en une seule fois, ou plus exactement de déclarer un *modèle* de classe, mais avec un (ou plusieurs) type paramétré qui pourra être changé à loisir. À chaque instanciation avec un nouveau type réel, c'est comme si vous écriviez toute une nouvelle classe, à partir du modèle, et donc rendez réelle une de ces classes potentielles.

Comme je l'ai dit, je ne m'étendrai pas sur le pourquoi du comment des génériques. Je vais immédiatement embrayer sur leur utilisation au quotidien. Si vous n'avez encore rien compris, ce n'est pas grave : tout va s'éclaircir petit à petit.

Vous pouvez aussi consulter le tutoriel  **Les génériques sous Delphi .NET** de Laurent Dardenne.

II - L'utilisation au quotidien : l'exemple TList<T>

Paradoxalement, nous allons commencer par voir comment utiliser une classe générique - c'est un abus de langage, il faudrait dire : modèle de classe générique - au quotidien, et non pas comment en écrire une. Il y a plusieurs bonnes raisons à cela.

D'une part, parce qu'il est beaucoup plus facile de concevoir et d'écrire une classe une fois que l'on a une idée assez précise de la façon dont on va l'utiliser. Et c'est encore plus vrai lorsqu'on découvre un nouveau paradigme de programmation.

D'autre part, la majorité des présentations de l'orienté objet en général explique d'abord comment se servir de classes existantes aussi.

II-A - Un code simple de base

Pour commencer en douceur, nous allons écrire un petit programme qui liste les carrés des nombres entiers de 0 à X, X étant défini par l'utilisateur.

Classiquement, on aurait utilisé un tableau dynamique (et ce serait bien mieux, rappelez-vous qu'on traite ici un cas d'école), mais on va utiliser une liste d'entiers.

Voici directement le code :

```
program TutoGeneriques;

{$APPTYPE CONSOLE}

uses
  SysUtils, Classes, Generics.Collections;

procedure WriteSquares(Max: Integer);
var
  List: TList<Integer>;
  I: Integer;
begin
  List := TList<Integer>.Create;
  try
    for I := 0 to Max do
      List.Add(I*I);

      for I := 0 to List.Count-1 do
        WriteLn(Format('%d*%0:d = %d', [I, List[I]]));
    finally
      List.Free;
    end;
  end;
end;

var
  Max: Integer;
begin
  try
    WriteLn('Entrez un nombre naturel :');
    ReadLn(Max);
    WriteSquares(Max);
  except
    on E:Exception do
      WriteLn(E.Classname, ': ', E.Message);
    end;
  end;

  ReadLn;
end.
```

Qu'est-ce qui est remarquable dans ce code ?

Tout d'abord, bien sûr, la déclaration de la variable **List** ainsi que la création de l'instance. Au nom du type **TList**, on a accolé le paramètre réel (ou assimilé : c'est un type, pas une valeur) entre chevrons < et >. (En anglais, on les appelle *angle brackets*, ce qui signifie littéralement : parenthèses en forme d'angles.)

Vous pouvez donc voir que, pour utiliser une classe générique, il est nécessaire, à chaque fois que vous écrivez son nom, de lui indiquer un type en paramètre. Pour l'instant, nous utiliserons toujours un type réel à cet endroit.

Certains langages avec génériques permettent ce qu'on appelle *l'inférence de type*, qui consiste, pour le compilateur, à deviner un type (ici on parle du paramètre de type). Ce n'est pas le cas de Delphi Win32. C'est pour ça que vous ne pouvez pas écrire :

```
var
  List: TList<Integer>;
begin
  List := TList.Create; // manque <Integer> ici
  ...
end;
```

La seconde chose est plus importante, et pourtant moins visible, puisqu'il s'agit d'une *absence*. En effet, aucun besoin de transtypage d'entier en pointeur et *vice versa* ! Pratique, non ? Et surtout, tellement plus lisible.

D'autre part, la sécurité de type est mieux gérée. Avec des transtypages, vous courez toujours le risque de vous tromper, et d'ajouter un pointeur à une liste censée contenir des entiers, ou l'inverse. Si vous utilisez correctement les génériques, vous n'aurez probablement presque plus jamais besoin de transtypage, et donc beaucoup moins d'occasion de faire des erreurs. De plus, si vous tentez d'ajouter un **Pointer** à un objet de classe **TList<Integer>**, c'est le compilateur qui vous enverra balader. Avant les génériques, une erreur de ce style pourrait n'avoir été révélée que par une valeur aberrante des mois après la sortie en production !

Notez que j'ai pris le type **Integer** pour limiter le code qui ne soit pas directement lié à la notion de généricité, mais on aurait pu utiliser *n'importe quel type* à la place.

II-B - Assignations entre classes génériques

Comme vous le savez, quand on parle d'héritage, on peut assigner une instance d'une classe fille à une variable d'une classe mère, mais pas l'inverse. Qu'en est-il avec la généricité ?

Partez d'abord du principe que vous ne pouvez rien faire ! Tous ces exemples sont *faux*, et ne compilent pas :

```
var
  NormalList: TList;
  IntList: TList<Integer>;
  ObjectList: TList<TObject>;
  ComponentList: TList<TComponent>;
begin
  ObjectList := IntList;
  ObjectList := NormalList;
  NormalList := ObjectList;
  ComponentList := ObjectList;
  ObjectList := ComponentList; // oui, même ça c'est faux
end;
```

Comme le commentaire le fait remarquer, ce n'est pas parce qu'un **TComponent** peut être assigné à un **TObject** qu'un **TList<TComponent>** peut être assigné à un **TList<TObject>**. Pour comprendre pourquoi, pensez que

TList<TObject>.Add(Value: TObject) permettrait, si l'affectation était valide, d'insérer une valeur de type **TObject** dans une liste de **TComponent** !

L'autre remarque importante consiste à préciser que **TList<T>** n'est en aucun cas une spécialisation de **TList**, ni une généralisation de celle-ci. En fait, ce sont deux types *totalemt différents*, le premier déclaré dans l'unité **Generics.Collections** et le second dans **Classes** !

Nous verrons plus loin dans ce tutoriel, qu'il est possible de faire certaines affectations grâce aux *contraintes* sur les paramètres génériques.

II-C - Les méthodes de TList<T>

Il est intéressant de constater que **TList<T>** ne propose pas le même ensemble de méthodes que **TList**. On trouve plus de méthodes de traitement "de plus haut niveau", et moins de méthodes "de bas niveau" (comme **Last**, qui a disparue).

Les nouvelles méthodes sont les suivantes :

- 3 versions surchargées de **AddRange**, **InsertRange** et **DeleteRange** : elles sont l'équivalent de **Add/Insert/Delete** respectivement pour une série d'éléments ;
- Une méthode **Contains** ;
- Une méthode **LastIndexOf** ;
- Une méthode **Reverse** ;
- 2 versions surchargées de **Sort** (le nom n'est pas nouveau, mais l'utilisation est toute autre) ;
- 2 versions surchargées de **BinarySearch**.

Si j'attire votre attention sur ces changements qui peuvent vous sembler anodins, c'est parce qu'ils sont très caractéristiques du changement dans la façon de concevoir qu'apportent les génériques.

Quel intérêt, en effet, y aurait-il eu à implémenter une méthode **AddRange** dans la désormais obsolète **TList** ? Aucun, car il aurait fallu transtyper chaque élément, tour à tour, donc il aurait fallu de toutes façons écrire une boucle pour construire le tableau à insérer. Alors autant appeler directement **Add** dans chaque tour de boucle.

Tandis qu'avec la généricité, il suffit d'écrire une fois le code, et il est vraiment valable, pleinement, pour tous les types.

Ce qu'il est important de remarquer et de comprendre ici, c'est que la généricité permet de factoriser beaucoup plus de comportements dans l'écriture d'une seule classe.

II-D - TList<T> et les comparateurs

Certes, **TList<T>** peut travailler avec n'importe quel type. Mais comment peut-elle savoir comment comparer deux éléments ? Comment savoir s'ils sont égaux, afin d'en rechercher un avec **IndexOf** ? Ou comment savoir si l'un est plus petit que l'autre, afin de trier la liste ?

La réponse vient des *comparateurs*. Un comparateur est une interface d'objet de type **IComparer<T>**. Eh oui, on reste en pleine généricité. Ce type est défini dans **Generics.Defaults.pas**

Lorsque vous créez une **TList<T>**, vous pouvez passer en paramètre au constructeur un comparateur, qui sera dès lors utilisé pour toutes les méthodes qui en ont besoin. Si vous ne le faites pas, un comparateur par défaut sera utilisé.

Le comparateur par défaut utilisé dépend du type des éléments, bien entendu. Pour l'obtenir, **TList<T>** appelle la méthode de classe **TComparer<T>.Default**. Cette méthode fait un travail un peu barbare à base de RTTI pour obtenir la meilleure solution dont il soit capable. Mais elle n'est pas toujours adaptée.

Vous pouvez conserver le comparateur par défaut pour les types de données suivants :

- Les types ordinaux (entiers, caractères, booléens, énumérations) ;
- Les types flottants ;
- Les types ensemble (uniquement pour l'égalité) ;
- Les types chaîne longue Unicode (**string**, UnicodeString et WideString) ;
- Les types chaîne longue ANSI (AnsiString) mais *sans code de page* pour les < et > ;
- Les types Variant (uniquement pour l'égalité) ;
- Les types classe (uniquement pour l'égalité - utilise la méthode **TObject.Equals**) ;
- Les types pointeur, meta-classe et interface (uniquement pour l'égalité) ;
- Les types tableau statique ou dynamique d'ordinaux, de flottants ou d'ensembles (uniquement pour l'égalité).

Pour tous les autres types, le comparateur par défaut fait une comparaison bête et méchante du contenu mémoire de la variable. Il vaut donc mieux alors écrire un comparateur personnalisé.

Pour ce faire, il existe deux moyens simples. L'un est basé sur l'écriture d'une fonction, l'autre sur la dérivation de la classe **TComparer<T>**. Nous allons les illustrer toutes les deux pour la comparaison de **TPoint**. Nous considérerons que ce qui classe les points est leur distance au centre - au point (0, 0) - afin d'avoir un ordre total (au sens mathématique du terme).

II-D-1 - Écrire un comparateur en dérivant TComparer<T>


Rien de plus simple, vous avez toujours fait ça ! Une seule méthode à surcharger : **Compare**. Elle doit renvoyer 0 en cas d'égalité, un nombre strictement positif si le paramètre de gauche est supérieur à celui de droite, et un nombre strictement négatif dans le cas contraire.

Voici ce que ça donne :

```
function DistanceToCenterSquare(const Point: TPoint): Integer; inline;
begin
    Result := Point.X*Point.X + Point.Y*Point.Y;
end;

type
    TPointComparer = class(TComparer<TPoint>)
        function Compare(const Left, Right: TPoint): Integer; override;
    end;

function TPointComparer.Compare(const Left, Right: TPoint): Integer;
begin
    Result := DistanceToCenterSquare(Left) - DistanceToCenterSquare(Right);
end;
```

 *Remarquez au passage l'héritage de **TPointComparer** : elle hérite de **TComparer<TPoint>**. Vous voyez donc qu'il est possible de faire hériter une classe "simple" d'une classe générique, pour peu qu'on lui fournisse un paramètre réel pour son paramètre générique.*

Pour utiliser notre comparateur, il suffit d'en créer une instance et de la passer au constructeur de la liste. Voici un petit programme qui crée 10 points au hasard, les trie et affiche la liste triée.

```
function DistanceToCenter(const Point: TPoint): Extended; inline;
begin
    Result := Sqrt(DistanceToCenterSquare(Point));
end;

procedure SortPointsWithTPointComparer;
```



```

const
    MaxX = 100;
    MaxY = 100;
    PointCount = 10;
var
    List: TList<TPoint>;
    I: Integer;
    Item: TPoint;
begin
    List := TList<TPoint>.Create(TPointComparer.Create);
    try
        for I := 0 to PointCount-1 do
            List.Add(Point(Random(2*MaxX+1) - MaxX, Random(2*MaxY+1) - MaxY));

            List.Sort; // utilise le comparateur passé au constructeur



            for Item in List do
                WriteLn(Format('%d'#9'%d'#9'(distance au centre = %.2f)',
                    [Item.X, Item.Y, DistanceToCenter(Item)]));
            finally
                List.Free;
            end;
        end;
    end;

begin
    try
        Randomize;

        SortPointsWithTPointComparer;
    except
        on E:Exception do
            WriteLn(E.Classname, ': ', E.Message);
        end;
    end;

    ReadLn;
end.

```

 Et où est la libération du comparateur instancié, dans tout ça ? Tout simplement dans le fait que **TList<T>** prend comme comparateur une interface de type **IComparer<T>**. Donc le comptage de références est appliqué (implémenté dans **TComparer<T>**). Ainsi, il n'est aucun besoin de se soucier de la vie, et donc de la libération, du comparateur. Si vous n'avez aucune idée du fonctionnement des interfaces sous Delphi, il vous sera très profitable de consulter le tutoriel  **Les interfaces d'objet sous Delphi** de Laurent Dardenne.

II-D-2 - Écrire un comparateur avec une simple fonction de comparaison

Cette alternative semble plus simple, d'après son nom : pas besoin de jouer avec des classes en plus. Pourtant, je la traite en second, car elle introduit un nouveau type de données disponible dans Delphi 2009. Il s'agit des *références* de routine.

Ah bon ? Vous connaissez ? Non, vous ne connaissez pas ;-) Ce que vous connaissez déjà, ce sont les *types procéduraux*, déclarés par exemple comme **TNotifyEvent** :

```

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

```

Les types référence de routine sont déclarés, dans cet exemple, comme **TComparison<T>** :

```

type

```

```
TComparison<T> = reference to function(const Left, Right: T): Integer;
```

Il y a au moins trois différences entre les types procéduraux et les types référence de routine.

La première est qu'un type référence de routine ne peut pas être marqué comme **of object**. Autrement dit, on ne peut jamais assigner une méthode à une référence de routine, seulement... Des routines. (Ou du moins, je n'ai pas encore réussi à le faire ^^.)

La deuxième est plus fondamentale : tandis qu'un type procédural (non **of object**) est un pointeur sur l'adresse de base d'une routine (son point d'entrée), un type référence de routine est en réalité une *interface* ! Avec comptage de références et ce genre de choses. Toutefois, vous n'aurez vraisemblablement jamais à vous en soucier, car son utilisation au quotidien est identique à celle d'un type procédural.

La dernière est ce qui explique l'apparition des références de routine. On peut assigner une routine anonyme - nous allons voir tout de suite à quoi ça ressemble - à une référence de routine, mais pas à un type procédural. Essayez, vous verrez que ça ne passe pas la compilation. Accessoirement, c'est aussi ce qui explique que les références de routine soient implémentées par des interfaces, mais la réflexion à ce sujet est hors du cadre de ce tutoriel.

Revenons à notre tri de points. Pour créer un comparateur sur base d'une fonction, on utilise une autre méthode de classe de **TComparer<T>** ; il s'agit de **Construct**. Cette méthode de classe prend en paramètre une référence de routine de type **TComparison<T>**. Comme déjà signalé, l'usage des références de routine est très similaire à celui des types procéduraux : on peut employer le nom de la routine comme paramètre, directement. Voici ce que ça donne :

```
function ComparePoints(const Left, Right: TPoint): Integer;
begin
    Result := DistanceToCenterSquare(Left) - DistanceToCenterSquare(Right);
end;

procedure SortPointsWithComparePoints;
const
    MaxX = 100;
    MaxY = 100;
    PointCount = 10;
var
    List: TList<TPoint>;
    I: Integer;
    Item: TPoint;
begin
    List := TList<TPoint>.Create(TComparer<TPoint>.Construct(ComparePoints));
    try
        for I := 0 to PointCount-1 do
            List.Add(Point(Random(2*MaxX+1) - MaxX, Random(2*MaxY+1) - MaxY));

            List.Sort; // utilise le comparateur passé au constructeur

        for Item in List do
            WriteLn(Format('%d'#9'%d'#9'(distance au centre = %.2f)',
                [Item.X, Item.Y, DistanceToCenter(Item)]));
        finally
            List.Free;
        end;
    end;

begin
    try
        Randomize;

        SortPointsWithComparePoints;
    except
        on E:Exception do
            WriteLn(E.Classname, ': ', E.Message);
        end;
    end;
end;
```

```
ReadLn;
end.
```

La seule différence provient donc, bien sûr, de la création du comparateur. Tout le reste de l'utilisation de la liste est identique (encore heureux !).

En interne, la méthode de classe **Construct** crée une instance de **TDelegatedComparer<T>**, qui prend en paramètre de son constructeur la référence de routine qui s'occupera de la comparaison. L'appel à **Construct** renvoie donc un objet de ce type, sous couvert de l'interface **IComparer<T>**.


Bon, c'était finalement très simple également. En fait, il faut bien s'en rendre compte : les génériques sont là pour nous faciliter la vie !

Mais j'ai lâché le morceau plus haut, on peut assigner une routine anonyme à une référence de routine. Alors voyons ce que cela pourrait donner :

```
procedure SortPointsWithAnonymous;
var
  List: TList<TPoint>;
  // ...
begin
  List := TList<TPoint>.Create(TComparer<TPoint>.Construct(
    function(const Left, Right: TPoint): Integer
    begin
      Result := DistanceToCenterSquare(Left) - DistanceToCenterSquare(Right);
    end));

  // Toujours la même suite...
end;
```

Cette forme de création est intéressante surtout si c'est le seul endroit où vous aurez besoin de la routine de comparaison.

 *En parlant des routines anonymes : oui, elles peuvent accéder à des variables locales de la routine/méthode englobante. Et oui, elle peuvent encore le faire après le retour de cette routine/méthode englobante. L'exemple suivant montre la chose :*

```
function MakeComparer(Reverse: Boolean = False): TComparison<TPoint>;
begin
  Result :=
    function(const Left, Right: TPoint): Integer
    begin
      Result := DistanceToCenterSquare(Left) - DistanceToCenterSquare(Right);
      if Reverse then
        Result := -Result;
      end;
    end;
end;

procedure SortPointsWithAnonymous;
var
  List: TList<TPoint>;
  // ...
begin
  List := TList<TPoint>.Create(TComparer<TPoint>.Construct(
    MakeComparer(True)));

  // Toujours la même suite...
end;
```

Intéressant, n'est-ce pas ?

Voilà qui clôt ce petit tour des comparateurs utilisés avec **TList<T>**, et avec lui cette introduction aux génériques à travers l'utilisation de cette classe. Dans le chapitre suivant, nous allons commencer à voir comment on peut écrire soi-même une classe générique.

III - Conception d'une classe générique

Maintenant que nous savons comment on utilise des classes génériques au quotidien, il est temps de voir comment fonctionne une classe générique à l'intérieur.

Pour cela, nous allons développer une classe **TTreeNode<T>**, qui sera une implémentation générique d'un arbre. Cette fois, il ne s'agit plus du tout d'un cas d'école. C'est une véritable classe que vous pourrez utiliser dans vos projets réels.

III-A - Déclaration de la classe

Commençons par le commencement : la déclaration de la classe. Comme vous avez pu l'apercevoir dans l'extrait de la déclaration de **TList<T>**, on indique un paramètre générique (traditionnellement **T** lorsqu'il n'a pas de signification précise) en chevrons. On va donc avoir :

```
unit Generics.Trees;

type
  TTreeNode<T> = class(TObject)
  end;
```

Un noeud d'un arbre sera étiqueté d'une valeur de type **T**, et aura de 0 à plusieurs enfants. Lorsqu'il est détruit, un noeud libère tous ses enfants.


Quand on dit une valeur de type **T**, eh bien, c'est simple, on déclare **FValue: T**; de la même manière qu'avec n'importe quel type normal. Pour garder la liste des enfants, on utilisera une **TObjectList<U>**, déclarée dans **Generics.Collections**. J'ai volontairement utilisé **U** ici, car il ne faut pas confondre. En fait, **U** sera **TTreeNode<T>** ! Eh oui, on peut utiliser comme paramètre générique réel une classe générique.

 *Nous n'implémentons pas ici un arbre de recherche ou de tri. En conséquence, nous n'avons pas besoin d'un comparateur comme **TList<T>**.*

En ce qui concerne les champs, cela va donc donner ceci :

```
type
  TTreeNode<T> = class(TObject)
  private
    FParent: TTreeNode<T>;
    FChildren: TObjectList<TTreeNode<T>>;
    FValue: T;
  end;
```

Étrange ? Si on y réfléchit bien, pas tant que ça. On a dit plus haut qu'un type générique pouvait être remplacé par n'importe quel type. Donc pourquoi pas un type classe générique ?

 *J'attire votre attention sur le fait que, dans le nom **TTreeNode<T>**, **T** est un paramètre générique non réel (formel, si vous préférez). Tandis qu'au sein de la classe, il devient un type réel ! C'est exactement comme les paramètres d'une routine. Dans la signature de la routine, ce sont des paramètres formels ; mais une fois dans le corps de la routine, ce sont des variables locales comme les autres. C'est pourquoi on peut utiliser **T** comme type de **FValue**, ou même comme type réel servant à paramétrer **TTreeNode<T>** dans la déclaration de **FChildren**.*

i Lorsque j'ai dit qu'on pouvait utiliser n'importe quel type comme paramètre réel, je vous ai mentis. Ce n'est pas toujours vrai. Par exemple, dans **TObjectList<T>**, **T** doit être un type classe. C'est parce que **TObjectList<T>** a posé une contrainte sur son paramètre de type. Nous verrons cela plus en détails dans un autre chapitre, mais je me dois de vous le signaler ici, puisque nous utilisons cette classe. Nous l'utilisons car nous voulons bénéficier de la libération automatique des objets contenus offerte par **TObjectList<T>**, ce que ne fait pas **TList<T>**.

Comme méthodes, outre les constructeur et destructeur, nous proposerons des méthodes de parcours en profondeur (préfixe et suffixe), ainsi que des méthodes d'ajout/déplacement/suppression de noeud. En fait, l'ajout sera demandé par l'enfant lui-même, lorsqu'on lui donnera un parent.

Pour les parcours, nous aurons besoin d'un type de call-back. Nous allons utiliser un type référence de routine. Et nous ferons deux versions de chaque parcours : un parcours sur les noeuds, et un parcours sur les valeurs des noeuds. Le second sera le plus souvent employé lors de l'utilisation de la classe **TTreeNode<T>**. Le premier est là pour permettre d'être plus général, et sera utilisé par plusieurs méthodes de **TTreeNode<T>** (notamment la seconde version du parcours).

Enfin, il y aura bien sûr des propriétés permettant d'accéder au parent, aux enfants et à la valeur étiquetée. Ce qui donne ceci. Vous pouvez voir qu'il n'y a rien de très nouveau, en dehors du fait qu'il y a plein de **T** partout.

Voici la déclaration complète de la classe - que, rassurez-vous, je vous donne après l'avoir complètement implémentée ^^.

```

type
    /// Référence à une routine de call-back avec un paramètre
    TValueCallBack<T> = reference to procedure(const Value: T);

    {*
     Structure arborescente générique
    *}
    TTreeNode<T> = class(TObject)
    private
        FParent: TTreeNode<T>;           /// Noeud parent (nil si racine)
        FChildren: TObjectList<TTreeNode<T>>; /// Liste des enfants
        FValue: T;                       /// Valeur étiquetée

        FDestroying: Boolean; /// Indique si est en train d'être détruit

    procedure DoAncestorChanged;

    function GetRootNode: TTreeNode<T>;
    function GetChildCount: Integer; inline;
    function GetChildren(Index: Integer): TTreeNode<T>; inline;
    function GetIndexAsChild: Integer; inline;

    function GetIsRoot: Boolean; inline;
    function GetIsLeaf: Boolean; inline;
    function GetDepth: Integer;

    protected
        procedure AncestorChanged; virtual;
        procedure Destroying; virtual;

        procedure AddChild(Index: Integer; Child: TTreeNode<T>); virtual;
        procedure RemoveChild(Child: TTreeNode<T>); virtual;

        procedure SetValue(const AValue: T); virtual;

    property IsDestroying: Boolean read FDestroying;
    public
        constructor Create(AParent: TTreeNode<T>; const AValue: T); overload;
        constructor Create(AParent: TTreeNode<T>); overload;
        constructor Create(const AValue: T); overload;
    
```

```

constructor Create; overload;
destructor Destroy; override;

procedure AfterConstruction; override;
procedure BeforeDestruction; override;

procedure MoveTo(NewParent: TTreeNode<T>; Index: Integer = -1); overload;
procedure MoveTo(Index: Integer); overload;

function IndexOf(Child: TTreeNode<T>): Integer; inline;

procedure PreOrderWalk(
    const Action: TValueCallback<TTreeNode<T>>; overload;
procedure PreOrderWalk(const Action: TValueCallback<T>); overload;


procedure PostOrderWalk(
    const Action: TValueCallback<TTreeNode<T>>; overload;
procedure PostOrderWalk(const Action: TValueCallback<T>); overload;

property Parent: TTreeNode<T> read FParent;
property RootNode: TTreeNode<T> read GetRootNode;
property ChildCount: Integer read GetChildCount;
property Children[Index: Integer]: TTreeNode<T> read GetChildren;
property IndexAsChild: Integer read GetIndexAsChild;

property IsRoot: Boolean read GetIsRoot;
property IsLeaf: Boolean read GetIsLeaf;

property Value: T read FValue write SetValue;
end;
    
```

Identifions ce qu'il y a encore de remarquable ici. En fait, pas grand chose. Si ce n'est qu'à chaque fois qu'un paramètre est de type **T**, il est déclaré **const**. En effet, nous ne savons pas à l'avance ce que pourra bien être **T** en vrai. Et il y a donc des possibilités pour que ce soit un type "lourd" (comme un **string** ou un **record**), pour lequel il est plus efficace d'utiliser **const**. Et comme **const** n'est jamais pénalisant (il est sans effet sur les types "légers", comme Integer), on le met toujours lorsqu'on travaille avec des types génériques.

 *La notion de type lourd ou léger est une notion qui m'est propre, et en rien officielle, donc les guillemets. J'entends par type lourd un type dont les paramètres gagnent (en rapidité d'exécution) à être passés comme **const** lorsque c'est possible. Il s'agit, de manière générale, des types qui s'étendent sur plus de 4 octets (hors flottants et Int64), et ceux qui nécessitent une initialisation. En (très) gros, on retiendra les chaînes, les **record**, les tableaux, interfaces (pas classes) et Variant.*

En revanche, lorsqu'un paramètre est de type **TTreeNode<T>**, on ne met pas **const**. En effet, quel que sera **T**, **TTreeNode<T>** restera un type classe, qui est un type léger.

III-B - Implémentation d'une méthode

Pour montrer les particularités de l'implémentation d'une méthode de classe générique, travaillons sur **GetRootNode**, qui est très simple. Elle s'écrit comme suit :

```

{ *
  Noeud racine de l'arbre
  @return Noeud racine de l'arbre
  *}
function TTreeNode<T>.GetRootNode: TTreeNode<T>;
begin
    if IsRoot then
        Result := Self
    else
        Result := Parent.RootNode;
end;
    
```

```
end;
```

La seule chose à prendre en considération ici est que, à *chaque* implémentation d'une méthode d'une classe générique, il faut respecifier le <T> à côté du nom de la classe. En effet, c'est nécessaire car **TTreeNode** pourrait très bien exister par ailleurs, et identifie donc un *autre* identificateur.

III-C - La pseudo-routine Default


Pour implémenter les deux constructeurs qui n'ont pas de paramètre **AValue**, il faudra initialiser **FValue** avec une valeur. Oui mais, comme on ne connaît pas le type de la valeur, comment écrire une valeur par défaut, et ce pour tous les types possibles ?

La solution est dans la nouvelle pseudo-routine **Default**. Tout comme **TypeInfo**, celle-ci prend en paramètre un identificateur de type. Et le nom d'un type générique est bien un identificateur de type.

Cette pseudo-routine "renvoie" la valeur par défaut du type demandé. On peut donc écrire :

```
{*
  Crée un noeud avec un parent mais sans valeur étiquetée
  @param AParent  Parent
  *}
constructor TTreeNode<T>.Create(AParent: TTreeNode<T>);
begin
  Create(AParent, Default(T));
end;

{*
  Crée un noeud sans parent ni valeur étiquetée
  *}
constructor TTreeNode<T>.Create;
begin
  Create(nil, Default(T));
end;
```

 *Dans le cas de l'initialisation d'un champ d'objet, ce n'est pas à proprement parler nécessaire. Car l'instanciation d'un objet initialise déjà tous ses champs à leur valeur par défaut. Mais je n'ai pas trouvé de meilleur endroit pour vous introduire à cette pseudo-routine.*

III-D - Les références de routine avec les parcours d'arbre

Afin d'illustrer l'utilisation des références de routine de l'autre côté du miroir, voici un commentaire sur l'implémentation des parcours d'arbre.

Chacune des deux surcharges du parcours prend en paramètre une référence de routine de rappel (communément appelée routine de call-back). Remarquez que, ici aussi, on a utilisé un paramètre **const**. En effet, le lecteur attentif se souviendra que les références de routine sont des interfaces, en réalité. Or une interface est un type lourd. Il faut donc utiliser **const** quand c'est possible.

Mise à part cette petite considération, il n'y a rien de spécial à dire sur la version parcours sur les noeuds, que voici :

```
{*
  Parcours préfixe sur les noeuds de l'arbre
  @param Action  Action à effectuer pour chaque noeud
  *}
constructor TTreeNode<T>.Create;
```



```

procedure TTreeNode<T>.PreOrderWalk(const Action: TValueCallBack<TTreeNode<T>>);
var
    Index: Integer;
begin
    Action(Self);
    for Index := 0 to ChildCount-1 do
        Children[Index].PreOrderWalk(Action);
end;
    
```


Pour appeler le call-back, on écrit exactement la même chose qu'avec un type procédural, à savoir la même forme qu'un appel de routine, mais avec la variable de type référence de routine en lieu et place du nom de la routine.

Pour implémenter la seconde version, on se sert de la première, en transmettant au paramètre **Action**... Une routine anonyme !

```

{ *
  Parcours préfixe sur les valeurs des noeuds de l'arbre
  @param Action  Action à effectuer pour chaque valeur
 * }
procedure TTreeNode<T>.PreOrderWalk(const Action: TValueCallBack<T>);
begin
    PreOrderWalk(
        procedure(const Node: TTreeNode<T>)
        begin
            Action(Node.Value);
        end;
    end;
    
```

C'est quand même élégant comme style de programmation, non ?

 Vous allez dire que je radotte avec mes **const**, mais bon quand même : le paramètre **Node** de la routine anonyme a été déclaré **const** alors qu'il est manifestement de type classe. C'est que : il faut quand même être compatible avec la signature du type **TValueCallBack<TTreeNode<T>>**, qui demande que son paramètre soit **const** ;-).

III-D-1 - D'autres méthodes qui utilisent les routines anonymes ?

Oui, deux autres. **DoAncestorChanged**, qui a pour mission de faire un parcours préfixe sur la méthode **AncestorChanged**. Et **BeforeDestruction**, qui doit faire un parcours préfixe sur la méthode **Destroying** de tous les enfants. C'est toujours la même chose :

```

{ *
  Appelle la procédure AncestorChanged sur toute la descendance (préfixe)
 * }
procedure TTreeNode<T>.DoAncestorChanged;
begin
    PreOrderWalk(
        procedure(const Node: TTreeNode<T>)
        begin
            Node.AncestorChanged;
        end;
    end;

    { *
      [@inheritDoc]
    * }
procedure TTreeNode<T>.BeforeDestruction;
begin
    inherited;
    
```

```

if not IsDestroying then
    PreOrderWalk(procedure (const Node: TTreeNode<T>) begin Node.Destroying end);

if (Parent <> nil) and (not Parent.IsDestroying) then
    Parent.RemoveChild(Self);
end;
    
```

III-E - Et le reste

Le reste, eh bien... Il est classique ;-). Rien de nouveau. Je ne vais donc pas m'éterniser dessus, mais le source complet est téléchargeable, comme tous les autres, **en fin de tutoriel**.

III-F - Comment utiliser TTreeNode<T> ?

C'est bien beau d'écrire une classe d'arbre générique, mais comment on l'utilise ?

Après une description en long et en large de la classe **TList<T>**, il n'y a pas grand chose à dire. Je vais donc juste vous donner le code d'un petit programme de test.

```

procedure TestGenericTree;
const
    NodeCount = 20;
var
    Tree, Node: TTreeNode<Integer>;
    I, J, MaxDepth, Depth: Integer;
begin
    Tree := TTreeNode<Integer>.Create(0);
    try
        // Créer l'arbre
        MaxDepth := 0;
        for I := 1 to NodeCount do
            begin
                Depth := Random(MaxDepth+1);
                Node := Tree;

                for J := 0 to Depth-1 do
                    begin
                        if Node.IsLeaf then
                            Break;
                        Node := Node.Children[Random(Node.ChildCount)];
                    end;

                    if TTreeNode<Integer>.Create(Node, I).Depth > MaxDepth then
                        Inc(MaxDepth);
                end;

                // Afficher l'arbre avec un parcours préfixe
                Tree.PreOrderWalk(
                    procedure (const Node: TTreeNode<Integer>)
                        begin
                            Write(StringOfChar(' ', 2*Node.Depth));
                            Write('- ');
                            WriteLn(Node.Value);
                        end);
            finally
                Tree.Free;
            end;
    end;
    
```

IV - Conception d'un record générique

Histoire d'aller voir d'un autre côté, développons le **record** simple mais utile **TNullable<T>**. Son but est de valoir soit une valeur de type **T**, soit **nil**. Il est fort probable que vous ayez déjà eu besoin d'un tel type, par exemple pour représenter la valeur NULL des bases de données.

Ce **record** contiendra deux champs : **FValue** de type **T** et **FIsNil** de type **Boolean**, ainsi que deux propriétés permettant de lire (mais pas d'écrire) ces champs. On se servira uniquement d'opérateurs de conversion implicite pour construire des valeurs de ce type.

```

unit Generics.Nullable;

interface

type
  TNullable<T> = record
  private
    FValue: T;
    FIsNil: Boolean;
  public
    class operator Implicit(const Value: T): TNullable<T>;
    class operator Implicit(Value: Pointer): TNullable<T>;
    class operator Implicit(const Value: TNullable<T>): T;

    property IsNil: Boolean read FIsNil;
    property Value: T read FValue;
  end;
    
```

 Pour plus d'infos sur les redéfinitions d'opérateurs, consultez le tutorial  **La surcharge d'opérateurs sous Delphi 2006 Win32** de Laurent Dardenne.

C'est donc un type *immuable* (dont on ne peut plus modifier l'état une fois qu'il est créé).

L'implémentation des trois opérateurs de conversion est assez simple. Le deuxième d'entre eux (celui avec un paramètre de type **Pointer**) est là pour permettre l'affectation `:= nil`.

```

uses
  SysUtils;

resourcestring
  sCantConvertNil = 'Ne peut convertir nil';
  sOnlyValidValueIsNil = 'La seule valeur valide est nil';

class operator TNullable<T>.Implicit(const Value: T): TNullable<T>;
begin
  Result.FValue := Value;
  Result.FIsNil := False;
end;

class operator TNullable<T>.Implicit(Value: Pointer): TNullable<T>;
begin
  Assert(Value = nil, sOnlyValidValueIsNil);
  Result.FIsNil := True;
end;

class operator TNullable<T>.Implicit(const Value: TNullable<T>): T;
begin
  if Value.IsNil then
    raise EConvertError.Create(sCantConvertNil);

  Result := Value.FValue;
    
```

```
end;
```

On peut l'utiliser très simplement comme ceci :

```
var  
  Value: Integer;  
  NullValue: TNullable<Integer>;  
begin  
  NullValue := 5;  
  WriteLn(Integer(NullValue));  
  NullValue := nil;  
  if NullValue.IsNil then  
    WriteLn('nil')  
  else  
    WriteLn(NullValue.Value);  
  
  NullValue := 10;  
  Value := NullValue;  
  WriteLn(Value);  
end;
```

Ce qui affiche bien :

```
5  
nil  
10
```

Vous avez tout compris ? C'est génial, parce que je n'ai donné aucune explication. C'est bien la preuve que c'est simple comme bonjour, les génériques :-).

Le code source complet de **Generics.Nullable** se trouve dans le zip des sources de ce tutoriel, téléchargeable **en fin de tutoriel**.

V - Contraintes sur les types génériques

Bien, vous connaissez maintenant les bases. Il est temps de passer aux choses sérieuses, à savoir : les contraintes.

Comme leur nom l'indique, les contraintes permettent d'imposer des restrictions sur les types réels qui peuvent être utilisés pour remplacer un paramètre formel de type. Pour continuer la comparaison avec les paramètres de méthode : une contrainte est au type ce que le type est à la variable paramètre. Pas clair ? Bon, lorsque vous spécifiez un type pour un paramètre, vous ne pouvez lui transmettre que des valeurs qui sont compatibles avec ce type. Lorsque vous spécifiez une contrainte sur un paramètre de type, vous devez le remplacer par un type réel qui satisfait ces contraintes.

V-A - Quelles sont les contraintes possibles ?

Il n'existe qu'un nombre restreint de contraintes possibles. En fait, il n'en existe que de trois types, dont un pour lequel je n'ai pas trouvé d'utilité :-s.

On peut donc restreindre un type générique à être :

- un type classe descendant d'une classe donnée ;
- un type interface descendant d'une interface donnée, ou un type classe implémentant cette interface ;
- un type ordinal, flottant ou **record** ;
- un type classe qui possède un constructeur sans argument.

Pour imposer une contrainte à un paramètre générique, on note :

```


type
  TStreamGenericType<T: TStream> = class
  end;

  TIntfListGenericType<T: IInterfaceList> = class
  end;

  TSimpleTypeGenericType<T: record> = class
  end;

  TConstructorGenericType<T: constructor> = class
  end;
    
```

Ce qui impose, respectivement, que **T** devra être remplacé par la classe **TStream** ou une de ses descendantes ; par **InterfaceList** ou une de ses descendante ; par un type ordinal, flottant ou **record** (un type de valeur non null, selon la terminologie de Delphi) ; ou enfin par un type classe qui possède un constructeur public sans argument.

 **<T: class>** doit être employé à la place de **<T: TObject>**... On comprend bien que **class** soit accepté mais on se demande pourquoi **TObject** est rejeté.

Il est possible de combiner plusieurs contraintes interface, ou une contrainte classe et une à plusieurs contraintes interface. Dans ce cas, le type réel utilisé doit satisfaire toutes les contraintes en même temps. La contrainte **constructor** peut également être combinée avec des contraintes classe et/ou interface. Il est même possible de combiner **record** avec une ou plusieurs contraintes interface, mais je ne vois pas comment un type pourrait bien satisfaire les deux en même temps (en .NET, c'est possible, mais, pour l'instant, pas en Win32) !



*Il se pourrait, mais ce n'est que pure spéculation de ma part, que ce soit en prévision d'une version future, dans laquelle le type **Integer**, par exemple, "implémenterait" l'interface **IComparable<Integer>**. Ce qui en ferait donc un type satisfaisant les deux contraintes d'une déclaration comme **<T: record, IComparable<T>>**.*

On peut aussi utiliser une classe ou interface générique comme contrainte, avec donc un paramètre à spécifier. Ce paramètre peut être le type **T** lui-même. Par exemple, on peut vouloir imposer que le type d'éléments doit pouvoir se comparer avec lui-même. On utilisera alors :

```
type
  TSortedList<T: IComparable<T>> = class(TObject)
  end;
```

Si l'on veut en plus que **T** soit un type classe, on peut combiner :

```
type
  TSortedList<T: class, IComparable<T>> = class(TObject)
  end;
```

V-B - Mais à quoi ça sert ?

« Je croyais que le but des génériques était justement d'écrire une fois le code pour tous les types. Quel est alors l'intérêt de *restreindre* les types possibles ? »

Eh bien cela permet au compilateur d'avoir plus d'informations sur le type utilisé. Par exemple, cela lui permet de savoir, avec un type `<T: class>`, qu'il est légitime d'appeler la méthode **Free** dessus. Ou avec un type `<T: IComparable<T>>`, qu'il est possible d'écrire **Left.CompareTo(Right)**.

Pour illustrer cela, nous allons créer une classe fille de **TTreeNode<T>**, **TObjectTreeNode<T: class>**. À l'instar de **TObjectList<T: class>** qui propose de libérer automatiquement ses éléments lorsque la liste est détruite, notre classe libérera sa valeur étiquetée lors de la destruction.

En fait, cela fait donc très peu de code, que je vais donner en une fois :

```
type
  { *
    Structure arborescente générique dont les valeurs sont des objets
    Lorsque le noeud est libéré, la valeur étiquetée est libérée également.
  * }
  TObjectTreeNode<T: class> = class(TTreeNode<T>)
  public
    destructor Destroy; override;
  end;

  {-----}
  { TObjectTreeNode<T> }
  {-----}

  { *
    [@inheritDoc]
  * }
  destructor TObjectTreeNode<T>.Destroy;
begin
  Value.Free;
  inherited;
end;
```

Voilà, c'est tout. Le but est uniquement de montrer la technique. Pas d'avoir un truc exceptionnel.

Il y a deux choses à constater ici. D'abord, on peut faire hériter une classe générique d'une autre classe générique, en réutilisant le paramètre générique (ou pas, d'ailleurs).

Ensuite, dans l'implémentation des méthodes d'une classe générique avec contraintes, les contraintes ne doivent pas (et ne peuvent pas) être répétées.

Vous pouvez par ailleurs supprimer la contrainte et tenter de compiler. Le compilateur vous arrêtera sur l'appel à **Free**. En effet, **Free** n'est pas disponible sur n'importe quel type. Mais sur n'importe quelle classe, bien.

V-C - Une variante avec constructor

Vous pourriez aussi vouloir que les deux constructeurs sans paramètre **AValue** de **TObjectTreeNode<T>** créent un objet pour **AValue** au lieu d'utiliser **Default(T)** (qui, au passage, renvoie **nil** ici car T est contraint à être une classe).

Vous pouvez, pour cela, utiliser la contrainte **constructor**, ce qui donne :

```

type
  { *
    Structure arborescente générique dont les valeurs sont des objets
    Lorsqu'un noeud est créé sans valeur étiquetée, une nouvelle valeur est
    créée avec le constructeur sans paramètre du type choisi.
    Lorsque le noeud est libéré, la valeur étiquetée est libérée également.
  * }
  TCreateObjectTreeNode<T: class, constructor> = class(TObjectTreeNode<T>)
  public
    constructor Create(AParent: TTreeNode<T>); overload;
    constructor Create; overload;
  end;

implementation

{ *
  [@inheritDoc]
  * }
  constructor TCreateObjectTreeNode<T>.Create (AParent: TTreeNode<T>);
  begin
    Create (AParent, T.Create);
  end;


{ *
  [@inheritDoc]
  * }
  constructor TCreateObjectTreeNode<T>.Create;
  begin
    Create (T.Create);
  end;
    
```

À nouveau, si vous retirez la contrainte **constructor** ici, le compilateur marquera une erreur sur le **T.Create**.

VI - Paramétrer une classe avec plusieurs types

Comme vous avez pu vous en douter, il est possible de paramétrer une classe avec plusieurs types. Chacun, éventuellement, avec ses contraintes.

Ainsi, la classe **TDictionary<TKey,TValue>** prend en paramètres deux types. Le premier est le type des clefs, le second le type des éléments. Cette classe implémente une table de hachage.

 *Ne vous y trompez pas : **TKey** et **TValue** sont bien des paramètres génériques (formels), pas des types réels. Ne vous laissez pas prendre au piège par la notation.*

La syntaxe de déclaration est un peu laxiste, sur ce point. Il est en effet possible de séparer les types par des virgules (,) ou par des point-virgule (;), éventuellement en mixant les deux quand il y a plus de deux types. Autant au niveau de la déclaration de la classe qu'au niveau de l'implémentation des méthodes. Par contre, au niveau de l'utilisation d'un type générique, vous devez utiliser des virgules !

Toutefois, si vous placez une ou plusieurs contrainte sur un type qui n'est pas le dernier de la liste, vous devrez utiliser un point-virgule pour le séparer du suivant. En effet, une virgule signifierait une seconde contrainte.

Aussi, permettez-moi de vous proposer une règle de style - qui n'est pas celle suivie par Embarcadero. Utilisez toujours des point-virgule dans la déclaration du type générique (là où vous êtes susceptible de pouvoir mettre des contraintes) et utilisez des virgules partout ailleurs (implémentation des méthodes, et utilisation du type).

Comme je n'ai pas de meilleur exemple de type générique à vous proposer que celle de **TDictionary<TKey,TValue>**, je vous conseille de consulter le code de cette classe (définie dans l'unité **Generics.Collections**, vous vous en doutiez probablement). En voici juste un extrait :

```


type
  TPair<TKey,TValue> = record
    Key: TKey;
    Value: TValue;
  end;

  // Hash table using linear probing
  TDictionary<TKey,TValue> = class(TEnumerable<TPair<TKey,TValue>>)
    // ...
  public
    // ...

    procedure Add(const Key: TKey; const Value: TValue);
    procedure Remove(const Key: TKey);
    procedure Clear;

    property Items[const Key: TKey]: TValue read GetItem write SetItem; default;
    property Count: Integer read FCount;

    // ...
  end;
    
```

 *Comme vous l'avez déjà remarqué, **TDictionary<TKey,TValue>** utilise des noms de types génériques plus explicites que les **T** que nous avons utilisé jusqu'ici. Vous devriez faire de même, à chaque fois que le type a une signification particulière, comme c'est le cas ici. Et d'autant plus lorsqu'il y a plus d'un type paramétré.*

VII - Autres types génériques

Jusqu'à présent, nous n'avons défini que des *classes* génériques. Pourtant, nous avons déjà rencontré des *interfaces* génériques (comme **IComparer<T>**) et vous venez de croiser un type *record* générique (**TPair<TKey,TValue>**).

Il est donc tout à fait possible de définir autant des interfaces ou des **record** génériques que des classes génériques. Il est également possible de déclarer un type tableau générique (statique ou dynamique) mais dont seul le type des *éléments* peut dépendre des types paramétrés ; mais il est peu probable que vous y trouviez une utilité réelle.

Il n'est donc pas possible de déclarer un type pointeur générique, ou un type ensemble générique :

```
type
  TGenericPointer<T> = ^T; // erreur de compilation
  TGenericSet<T> = set of T; // erreur de compilation
```

VIII - Méthodes génériques

Nous avons exploré jusqu'ici les différentes possibilités offertes par les génériques sur les types définis par le développeur. Mais il est également possible d'écrire des *méthodes* génériques.


Dans beaucoup de présentations des génériques ou des templates pour d'autres langages, cette forme d'utilisation des génériques est présentée en premier. Mais encore une fois, j'ai préféré vous présenter d'abord ce qui sert souvent avant de m'intéresser aux utilisations moins fréquentes des génériques.


VIII-A - Une fonction Min générique


Pour présenter le concept, nous allons écrire une méthode de classe **TArrayUtils.Min<T>**, qui trouve et renvoie le plus petit élément d'un tableau. Nous aurons donc besoin d'utiliser un comparateur de type **IComparer<T>**.

Tout comme le nom du type devait l'être, le nom de la méthode doit être suivi des paramètres génériques entre chevrons. Ici le type générique est le type des éléments du tableau.

```
type
  TArrayUtils = class
  public
    class function Min<T>(const Items: array of T;
      const Comparer: IComparer<T>): T; static;
  end;
```

 *Et non ! Il n'est pas possible de déclarer une routine globale avec des paramètres génériques. Une raison possible viendrait du parallélisme avec la syntaxe de Delphi.NET, afin de réduire les coûts de développement et de maintenance, en interne.*

 *Pour pallier à ce manque, on utilise donc des méthodes de classe. Et mieux, on la précise comme étant **static**. Pas grand chose à voir avec le mot-clef du même nom en C++. Il s'agit ici de liaison statique. C'est-à-dire que dans une telle méthode, il n'y a pas de **Self**, et que donc l'appel à des méthodes de classe virtuelles, ou à des construteurs virtuels, n'est pas "virtualisé". Autrement dit, c'est comme s'il n'était pas virtuel. Au final, cela fait de la méthode de classe statique une authentique routine globale, mais avec un espace de noms différent.*

 *Contrairement à certains langages, il n'est pas nécessaire qu'un paramètre au moins reprenne chaque type générique introduit pour la méthode. Ainsi, il est permis d'écrire une méthode **Dummy<T>(Int: Integer): Integer**, qui n'a donc aucun paramètre formel dont le type est le paramètre générique **T**. En C++, par exemple, ça ne passerait pas.*

Côté implémentation de la méthode, c'est tout à fait similaire aux classes. Il faut répéter les chevrons et les noms des types génériques, mais pas leurs contraintes éventuelles. Cela donne donc :

```
class function TArrayUtils.Min<T>(const Items: array of T;
  const Comparer: IComparer<T>): T;
var
  I: Integer;
begin
  if Length(Items) = 0 then
    raise Exception.Create('No items in the array');

  Result := Items[Low(Items)];
  for I := Low(Items)+1 to High(Items) do
    if Comparer.Compare(Items[I], Result) < 0 then
```

```

Result := Items[I];
end;
    
```

Rien de bien exceptionnel donc ;-)

VIII-B - La surcharge et les contraintes

Profitions de ce bel exemple pour revoir nos contraintes, et proposer une version surchargée pour les types d'éléments qui supportent l'interface **IComparable<T>** (cette interface est définie dans **System.pas**).

Et avant ça, ajoutons une autre version surchargée qui prend une référence de routine de type **TComparison<T>**. Rappelez-vous qu'on peut facilement "transformer" un call-back **TComparison<T>** en une interface **IComparer<T>** avec **TComparer<T>.Construct**.


Vous pouvez observer l'utilisation de la méthode **CompareTo** sur le paramètre **Left**. Ceci n'est bien sûr possible que parce que, dans cette surcharge, le type **T** est contraint à supporter l'interface **IComparable<T>**.

```

type
TArrayUtils = class
public
    class function Min<T>(const Items: array of T;
        const Comparer: IComparer<T>): T; overload; static;
    class function Min<T>(const Items: array of T;
        const Comparison: TComparison<T>): T; overload; static;
    class function Min<T: IComparable<T>>(
        const Items: array of T): T; overload; static;
end;

class function TArrayUtils.Min<T>(const Items: array of T;
    const Comparison: TComparison<T>): T;
begin
    Result := Min<T>(Items, TComparer<T>.Construct(Comparison));
end;

class function TArrayUtils.Min<T>(const Items: array of T): T;
var
    Comparison: TComparison<T>;
begin
    Comparison :=
        function(const Left, Right: T): Integer
        begin
            Result := Left.CompareTo(Right);
        end;
    Result := Min<T>(Items, Comparison);
end;
    
```

 **Remarquez l'appel à `Min<T>` : Il est indispensable de spécifier à l'appel aussi le (ou les) type réel utilisé. Ceci contraste avec d'autres langages comme le C++.**

Maintenant, nous voulons proposer une quatrième version surchargée, toujours avec uniquement le paramètre **Items**, mais avec un paramètre **T** non contraint. Cette version devrait utiliser **TComparer<T>.Default**.

Mais ceci n'est pas possible ! Car, bien que les contraintes sur les types changent, les *paramètres* (arguments) sont les mêmes. Donc les deux versions surchargées sont totalement ambiguës ! Ainsi, la déclaration supplémentaire suivante échouera à la compilation :

```

type
TArrayUtils = class
    
```

```

public
  class function Min<T>(const Items: array of T;
    const Comparer: IComparer<T>): T; overload; static;
  class function Min<T>(const Items: array of T;
    const Comparison: TComparison<T>): T; overload; static;
  class function Min<T: IComparable<T>>(
    const Items: array of T): T; overload; static;
  class function Min<T>(
    const Items: array of T): T; overload; static; // Erreur de compilation
end;
    
```

Il faut donc faire un choix : abandonner l'un ou l'autre, ou utiliser un autre nom. Et comme, jusqu'à ce que les types de base comme **Integer** supportent l'interface **IComparable<T>**, vous risquez d'utiliser aussi souvent l'un que l'autre, il va falloir opter pour l'autre nom ;-)

```

type
  TArrayUtils = class
  public
    class function Min<T>(const Items: array of T;
      const Comparer: IComparer<T>): T; overload; static;
    class function Min<T>(const Items: array of T;
      const Comparison: TComparison<T>): T; overload; static;
    class function Min<T: IComparable<T>>(
      const Items: array of T): T; overload; static;

    class function MinDefault<T>(
      const Items: array of T): T; static;
  end;

  class function TArrayUtils.MinDefault<T>(const Items: array of T): T;
begin
  Result := Min<T>(Items, IComparer<T>(TComparer<T>.Default));
end;
    
```

Pourquoi le transtypage explicite en **IComparer<T>** de **Default** qui est pourtant manifestement *déjà* un **IComparer<T>** ? Parce que les références de routine et les surcharges ne font pas encore très bon ménage, et le compilateur a l'air de s'emmêler les pincesaux. Sans le transtypage, la compilation ne passe pas...

VIII-C - Des ajouts pour TList<T>

Si la classe **TList<T>** est une belle innovation, il n'en demeure pas moins qu'elle pourrait contenir plus de méthodes d'intérêt pratique.

Voici donc par exemple une implémentation de la méthode .NET **FindAll** pour **TList<T>**. Cette méthode a pour but de sélectionner une sous-liste à partir d'une fonction prédicat. Ce qu'on appelle fonction prédicat est une routine de call-back qui prend en paramètre un élément de la liste, et renvoie **True** s'il faut le sélectionner. On définit donc un type référence de routine **TPredicate<T>** comme suit :

```

unit Generics.CollectionsEx;

interface

uses
  Generics.Collections;

type
  TPredicate<T> = reference to function(const Value: T): Boolean;
    
```

Ensuite, comme malheureusement il semble impossible d'écrire un **class helper** pour une classe générique, nous allons écrire une méthode de classe **FindAll<T>** qui va faire cela. Puisqu'on est privé de **class helper**, on va en moins en profiter pour être plus général, et travailler sur un énumérateur quelconque, avec une surcharge pour un énumérable quelconque.

```

type
  TListEx = class
  public
    class procedure FindAll<T>(Source: TEnumerator<T>; Dest: TList<T>;
      const Predicate: TPredicate<T>); overload; static;
    class procedure FindAll<T>(Source: TEnumerable<T>; Dest: TList<T>;
      const Predicate: TPredicate<T>); overload; static;
  end;

implementation

class procedure TListEx.FindAll<T>(Source: TEnumerator<T>; Dest: TList<T>;
  const Predicate: TPredicate<T>);
begin
  while Source.MoveNext do
  begin
    if Predicate(Source.Current) then
      Dest.Add(Source.Current);
  end;
end;

class procedure TListEx.FindAll<T>(Source: TEnumerable<T>; Dest: TList<T>;
  const Predicate: TPredicate<T>);
begin
  FindAll<T>(Source.GetEnumerator, Dest, Predicate);
end;


end.
    
```

On peut s'en servir comme ceci :

```

Source := TList<Integer>.Create;
try
  Source.AddRange([2, -9, -5, 50, 4, -3, 7]);
  Dest := TList<Integer>.Create;
  try
    TListEx.FindAll<Integer>(Source, Dest, TPredicate<Integer>(
      function(const Value: Integer): Boolean
      begin
        Result := Value > 0;
      end)));

    for Value in Dest do
      WriteLn(Value);
  finally
    Dest.Free;
  end;
finally
  Source.Free;
end;
    
```

 *À nouveau, le transtypage est nécessaire à cause des surcharges. C'est assez déplorable, mais c'est comme ça. Si vous préférez ne pas avoir de transtypage, utilisez des noms différents, ou débarrassez-vous d'une des deux versions.*

Il ne tient qu'à vous de compléter cette classe avec d'autres méthodes de ce genre :-)

IX - RTTI et génériques

En dernier chapitre de ce tutoriel, voici quelques informations sur ce que deviennent les RTTI avec les génériques. Si vous ne jouez jamais avec les RTTI, vous pouvez sauter entièrement ce chapitre. Ce n'est pas du tout une introduction aux RTTI.

IX-A - Les changements sur la pseudo-routine TypeInfo

Les RTTI, ça commence toujours par la pseudo-routine **TypeInfo**. Vous savez peut-être qu'on ne peut pas appeler cette pseudo-routine sur n'importe quel type ; exemple : les types pointeur. Et que, de ce fait, elle ne renvoie jamais **nil**.

Alors, peut-on appeler **TypeInfo** sur un type générique **T** ? La question est pertinente : **T** pourrait bien être un type pointeur (invalide pour **TypeInfo**), mais également un type entier par exemple (valide pour **TypeInfo**).

La réponse est *oui*, on peut appeler **TypeInfo** sur un type générique. Mais que se passe-t-il alors si **T** se trouve être un type qui n'a pas de RTTI ? Eh bien, dans ce cas, et dans ce cas seulement, **TypeInfo** renvoie **nil**.

Pour illustrer la chose, voici une petite méthode de classe qui affiche le nom et la sorte d'un type mais *via* des génériques :

```

type
  TRTTI = class(TObject)
  public
    class procedure PrintType<T>; static;
  end;

class procedure TRTTI.PrintType<T>;
var
  Info: PTypeInfo;
begin
  Info := TypeInfo(T); // attention ! Info peut valoir nil ici

  if Info = nil then
    WriteLn('Ce type ne possède pas de RTTI')
  else
    WriteLn(Info.Name, #9, GetEnumName(TypeInfo(TTypeKind), Byte(Info.Kind)));
end;
    
```

On l'utilise de manière un peu particulière, dans le sens où le véritable paramètre de la routine **PrintType** est transmis en tant que type paramétré.

```

begin
  TRTTI.PrintType<Integer>;
  TRTTI.PrintType<TObject>;
  TRTTI.PrintType<Pointer>;
end;
    
```

Ce qui donne :

```

Integer tkInteger
TObject tkClass
Ce type ne possède pas de RTTI
    
```

IX-A-1 - Une fonction TypeInfo plus générale

Il m'est déjà arrivé de regretter que **TypeInfo** ne puisse pas être appelée sur n'importe quel type, quitte à recevoir **nil** ; peut-être que vous aussi. Voici donc une petite méthode de remplacement qui fait ça, à base de génériques.

```

type
  TRTTI = class(TObject)
  public
    class procedure PrintType<T>; static;
    class function TypeInfo<T>: PTypeInfo; static;
  end;

class function TRTTI.TypeInfo<T>: PTypeInfo;
begin
  Result := System.TypeInfo(T);
end;
    
```

Que vous pouvez utiliser comme ceci :

```

Info := TRTTI.TypeInfo<Pointer>; // Info = nil
// au lieu de :
Info := TypeInfo(Pointer); // erreur ici car Pointer n'a pas de RTTI
    
```

IX-B - Les types génériques ont-ils des RTTI ?

Il y a deux questions à se poser : est-ce que les types génériques *non instanciés* (avec donc un paramètre **T** non défini) ont des RTTI ? Et est-ce que les types génériques *instanciés* (donc où **T** a été remplacé par un type réel, comme **Integer**) ont des RTTI ?

Le plus facile pour le savoir est de l'essayer ;-). Vous pourrez observer, en testant, que seuls les types génériques *instanciés* ont des RTTI. En fait, c'est assez logique, dans la mesure où les types génériques non instanciés ne sont pas réellement des types, mais des *modèles* de type, et n'existe tout simplement plus du tout une fois passée la compilation.

On peut se demander quel *nom* on va trouver pour de tels types. Alors voici :

```


begin
  TRTTI.PrintType<TComparison<Integer>>;
  TRTTI.PrintType<TTreeNode<Integer>>;
end;
    
```

Ce qui donne :

```

TComparison<System.Integer>  tkInterface
TTreeNode<System.Integer>    tkClass
    
```

Ce qui montre que le nom comprend, entre chevrons, le nom complètement qualifié du type réel remplaçant le type générique.

 Vous pouvez aussi remarquer qu'on obtient bien **tkInterface** pour le type référence de routine **TComparison<T>**, ce qui prouve bien que c'est une interface.

Voilà, ce sont les seuls changements apportés aux RTTI avec l'avènement des génériques. Je ne parle bien sûr pas d'autres changements apportés dans cette version mais qui concernent les chaînes Unicode.

X - Conclusion

Voilà qui met fin à cette découverte des types génériques avec Delphi 2009. Si vous n'avez jamais eu l'expérience des génériques dans un autre langage, cela peut paraître déroutant. Mais ils sont réellement très pratiques, et changent la vie du développeur, dès qu'on joue bien avec.

XI - Télécharger les sources

Toutes les sources de ce tutoriel se trouvent dans le zip [Source sources.zip](#) ([Source miroir HTTP](#)).

XII - Remerciements

Un grand merci à **Laurent Dardenne** et à **SpiceGuid** pour leurs nombreux commentaires et corrections, qui ont permis d'augmenter la qualité de ce tutoriel.