Les génériques sous Delphi .NET

par Laurent Dardenne (Contributions) >

Date de publication : 23/10/2007

Dernière mise à jour : 23/10/2007

Delphi 2007 alias Highlander propose comme nouveautés du langage la prise en charge des génériques. Voyons dans le détail comment les utiliser.
Je tiens à remercier **Giovanny Temgoua** pour ses corrections orthographiques.

1 - Public concerné

Testé sous le Framework .NET 2.0 et Delphi 2007 beta.

Un rappel sur **les nouveautés du framework dotNET 2.0**.

1-1 - Les sources

Les fichiers sources des différents exemples :

FTP.

HTTP.

2 - Les génériques qu'est-ce que c'est ?

Le principe même des génériques est la réutilisation de traitements ou de structures de données sans se soucier du type de donnée manipulé. On utilisera évidemment le plus souvent les génériques sur des notions générales par exemple une liste ou une pile et moins sur une notion particulière tel qu'un tableau des jours de la semaine.

Les génériques traduisent le concept du **polymorphisme paramétré**.

3 - A quoi servent-ils

Prenons le cas de figure d'une structure de données telle qu'une pile, elle peut contenir des entiers, des objets ou tout autre type reconnus.

Aujourd'hui sous Win32 ou dotNET 1.1 il nous faut déclarer un type **TStack** pour chaque type de donnée existant ou tout du moins utilisé dans nos traitements. Avec Delphi 2007 et uniquement sous dotNET 2.0 il est désormais possible de définir une fois pour toute une classe **TStack** générique, ce qui nous permettra de préciser lors de son utilisation quel type on souhaite manipuler. La généricité nous offrant ici la construction de **type paramétrable** et permet donc de faire abstraction lors de la conception des types de données manipulés.

Sous .NET 2.0 l'usage des génériques supprime le *boxing/unboxing* et les opérations de transtypage entre les types de données, limitant ainsi les vérifications de type lors de l'exécution du programme.

Voir sur le forum Delphi cet **exemple** propice à l'usage de type générique.

Voici un extrait commenté de l'unité **ActnMenus** :

```
    //Pile spécialisée afin de manipuler le type TCustomActionMenuBar;
  TMenuStack = class(TStack)
  ...
function TMenuStack.GetBars(const Index: Integer): TCustomActionMenuBar;
begin
   //Transtypage obligatoire car
   //l'appel de TList.Get renvoi un objet.
  Result := TCustomActionMenuBar(List[Index]);
end;

//La pile est construite autour d'une liste d'objet
function TList.Get(Index: Integer): TObject;
begin
  Result := FList[Index];
end;

 //FList étant déclarée ainsi :
 property List: System.Collections.ArrayList read FList;
```

*Le framework 2.0 propose dans l'espace de nom **System.Collections.Generic** des collections et des interfaces génériques.*

*

4 - Comment ça fonctionne ?

Projet : ..\PremierGenerique

Etudions la déclaration d'une classe générique :

```
type
  TGenericType<AnyType> = class
    FData: AnyType;
    function GetData: AnyType;
    procedure SetData(Value: AnyType);
    property Data: AnyType read GetData write SetData;
  end;

function TGenericType<AnyType>.GetData: AnyType;
begin
  Result := FData;
end;

procedure TGenericType<AnyType>.SetData(Value: AnyType);
begin
  FData := Value;
end;
```

Ici **TGenericType<AnyType>** indique une classe générique, **AnyType** étant le paramètre de type référençant un type de donnée tel qu'***Integer*** ou ***String***. Chaque occurrence de **AnyType** présente dans cette déclaration sera, lors de la construction du type à la compilation, substituée par l'argument de type que l'on précisera de cette manière :

```
type
  TGenericTypeInt = TGenericType<Integer>; //Déclare un type à partir d'une classe générique
```

TGenericType<Integer> est appelé un *type construit fermé*. **Integer** étant appelé ici un argument de type pour **AnyType**.

TGenericType<AnyType> étant quant à lui un *type construit ouvert* tout comme **TGenericAutre<V,String>**, car **V** est un type indéterminé.

Notez que le nom du paramètre de type, utilisé dans une déclaration de générique, ne défini pas l'unicité du type :

```
type
  TGenericType<AnyType> = class
    FData: AnyType;
  end;

  TGenericType<ParametreDeType> = class //Erreur à la compilation
    FData: ParametreDeType;
  end;

  TGenericType<AnyType,N> = class //Surcharge Autorisée
    FData1: AnyType;
    FData2: N;
```

```
    end;
```

La présence de la classe générique **TGenericType<ParametreDeType>** provoquera, à la compilation, l'erreur suivante :

```
E2037 : La déclaration de 'TGenericType<AnyType>' diffère de la déclaration précédente
```

Vous remarquerez qu'une déclaration de générique peut contenir plusieurs paramètres de type séparés par une virgule. Comme par exemple la classe du framework 2.0 ***Dictionary****.*

Les limites actuelles sous Delphi :

- le débogueur n'évalue pas les génériques.

- le refactoring ne prend pas en charge les génériques.

- error insight ne reconnait pas les génériques.

5 - Un exemple détaillé

Après avoir déclaré notre classe et ses membres, voyons la création d'une instance de notre classe paramétrée. Essayons la déclaration de variable suivante :

```
var
  I: TGenericType;
```

ici la compilation échoue en provoquant l'erreur :

```
 Identificateur non déclaré : ' TGenericType ' (E2003).
```

Pour instancier un générique il nous faut au préalable déclarer un type construit fermé qui précise l'usage de notre classe paramétrable :

```
type
  TGenericTypeInt = TGenericType<Integer>; // La classe générique est paramétrée avec le type
 Integer
```

Ensuite l'appel de son constructeur reste classique :

```
var
  I: TGenericTypeInt;
begin
  I := TGenericTypeInt.Create;
```

Les constructions suivantes restent possibles :

```
var X: TGenericType<Integer>;

begin
  X := TGenericType<Integer>.Create;
  X.Data := 100;

  With TGenericType<Integer>.Create do
   Data := 100;
```

Sachez que les variables *X* et *I* sont compatibles en affectation.

Construisons un second type à partir de notre classe générique :

Projet : ..\PremierGenerique2

```
type
  TGenericTypeString = TGenericType<string>;
var
  I: TGenericTypeInt;
  D: TGenericTypeString;
```

```
begin
  I := TGenericTypeInt.Create;
  I.Data := 100;
  WriteLn(I.Data);

  D := TGenericTypeString.Create;
  D.Data := 'Generic';
  WriteLn(D.Data);
  I:=D;  //Erreur à la compilation
```

La dernière affectation, ***I:=D***, provoque l'erreur suivante :

```
E2010 Types incompatibles : 'TGenericType<System.Int32>' et 'TGenericType<System.String>'
```

L'affectation entre différents types, construits à partir d'une même classe générique, n'est pas possible cela est dû au typage fort des génériques.

6 - Les types de bases supportant les génériques

Certains types peuvent être à la base d'une déclaration de générique, en voici la liste.

6-1 - Les classes

Le type **class**, de type référence, pourra être dérivé.

```
 //Classe générique
TGenericClass<C>=Class
 FData: T;
End;
```

De plus, n'importe quelle classe imbriquée dans une déclaration générique de classe ou une déclaration générique de record est elle-même une déclaration générique de classe, puisque le paramètre de type pour le type contenant doit être assuré pour créer un type construit.

Les exceptions étant des classes il est tout à fait possible de créer des exceptions génériques.

6-2 - Les enregistrements

Le type **record**, de type valeur, ne pourra être dérivé car il ne supporte pas l'héritage.

```
 //Record générique
TGenericEnregistrement<T>=Record
 Data: T;
End;
```

6-3 - Les tableaux

La définition de tableaux permet l'usage de paramètre de type.

```
type
 TGeneriqueArray<X> = array of X;
 TGeneriqueArray2D<Y> = array of TGeneriqueArray<Y>; // Equivalent à Array of Array of Y

 //Tableau à une dimension contenant des chaînes de caractères
TGtabString  = TGeneriqueArray<String>;
 //Tableau à deux dimensions contenant des entiers
TGtab2DInt   = TGeneriqueArray2D<Integer>;

 //Types compatibles
TGtabInteger=TGeneriqueArray<Integer>;
TGtabIntegerV2 = array of Integer;
```

Notez que les deux derniers types sont compatibles.

6-4 - Les procédures et fonctions

Projet : ..\ProcedureEtFonction

Les paramètres et le type de retour d'une fonction peuvent être des paramètres de type.

```
type
  TProcedureGenerique<A> = procedure(Param1 : A);
  TProcObjetGenerique<B> = procedure(X,Y: B) of object; //Méthode d'objet
  TFonctionGenerique<T>  = Function : T;

  TMaClasse = class
    procedure UneMethode<T>(X, Y: T); // Même signature que le type TProcObjetGenerique<B>
    procedure TestMethode;
    procedure TestProcedure<UnType>(Prc:TProcedureGenerique<UnType>);
    procedure TestFonction<T>(fnct: TFonctionGenerique<T>);
  end;
```

L'usage d'un paramètre de type dans la signature de la procédure **TestProcedure** permet de le propager afin de déclarer le type de l'argument nommé *Prc*. Sinon la présence seul d'un argument de type dans la signature d'une méthode

```
  TMaClasse = class
    ...
    procedure TestProcedure(Prc:TProcedureGenerique<UnType>);
  end;
```

signalerait un identificateur inconnu pour *UnType*.

Ici on pourra donc passer en paramètre n'importe quelle procédure de type **TProcedureGenerique<A>**.

Voici un exemple d'appel :

```
Procedure ProcedureGeneriqueInt(M:Integer);
begin
  Writeln(M);
end;

Procedure ProcedureGeneriqueString(M:String);
begin
  Writeln(M);
end;

..
  With TMaClasse.Create do
  begin
   TestProcedure<Integer>(ProcedureGeneriqueInt);

   //TestProcedure<Integer>(ProcedureGeneriqueString); //E2010 Types incompatibles : 'Integer' et
 'string'
   //TestProcedure<String>(ProcedureGeneriqueInt); //E2010 Types incompatibles : 'string' et
 'Integer'
   TestProcedure<String>(ProcedureGeneriqueString);
  end;
```

Notez que l'on doit préciser un argument de type pour ce type d'appel.

La construction suivante ***Procedure ProcedureGenerique<A>(M:A);*** *provoque l'erreur suivante à la compilation :*

E2530 Les paramètres de type ne sont pas autorisés sur la fonction ou la procédure globale.

Ce qui signifie que l'usage d'un type construit ouvert ne pourra donc se faire qu'au travers d'un membre d'une classe ou d'un record.

Voyons maintenant l'appel d'une méthode :

```
procedure TMaClasse.UneMethode<T>(X, Y: T);
begin
  Writeln(X.ToString,' , ',Y.ToString);
end;

procedure TMaClasse.TestMethode;
var
  P: TProcObjetGenerique<Boolean>; //Argument de type Boolean connu
begin
  UneMethode<String>('Hello', 'World'); //Argument de type String connu
  UneMethode('Hello', 'World');

  UneMethode<Integer>(10, 20); //Argument de type Integer connu
  UneMethode(10, 20);

  P:=UneMethode<Boolean>;
  P(False,True);
end;
```

Les différents arguments type de notre méthode générique étant précisés dans le corps de cette méthode de test, sa manipulation ne pose pas de problème particulier.

6-4-1 - Le mot clé Default

Si vous avez chargé et exécuté le code du projet précédent vous aurez noté le déclenchement d'une exception ***NullReferenceException*** lors de l'appel de procédure **TestProcedure<String>** ainsi que l'usage du mot clé **Default**.

Regardons de plus près cette instruction.

```
procedure TMaClasse.TestProcedure<UnType>(Prc:TProcedureGenerique<UnType>);
var
  P: TProcedureGenerique<UnType>;
  Value: UnType;
begin
   //Le type est déterminé dans la signature, il est donc inconnu.
  Prc('Hello');
  P:=Prc;
  //Value:='Chaine'; //E2010 Types incompatibles : 'UnType' et 'string'
  Value:=Default(UnType);

  P(True);
  P(Default(UnType));
end;
```

Dans la signature et le corps de la procédure générique précédente, le type réel de **UnType** est inconnu au moment de l'écriture de ce code. On ne peut donc assigner une valeur quelconque à la variable *Value*.

Default renvoi la valeur par défaut du type, précisé par l'argument type utilisé. Pour le type *Integer* **Default** renverra 0.

Integer étant un type valeur, la valeur par défaut est dans ce cas connue mais si le type utilisé est un type référence, comme un **TObjet** ou une *String*, la valeur renvoyée sera égale à **Nil**.

⚠ *On testera donc, avant toute manipulation, le contenu des variables de type référence en utilisant l'instruction **Assigned**.*

```
if assigned(TObject(Value)) //Le cast est obligatoire
  then writeln('Assigné.')
  else writeln('Non assigné.');
```

Le type étant inconnu lors de la compilation la manipulation de la variable **Value** nécessite un transtypage en **TObject** et donc une opération de boxing.

6-5 - Les méthodes

Projet : ..\Methodes1

Revenons sur la manipulation des méthodes génériques, dans la procédure **TMaClasse.TestMethode** vous aurez remarqué 2 types d'appel pour **UneMethode<String>** :

```
UneMethode<String>('Hello', 'World');
UneMethode('Hello', 'World');
```

Le second appel ne précise pas l'argument de type car on utilise **l'inférence de type** qui n'est autre que la déduction de l'argument de type par le compilateur d'après le type de donnée des arguments utilisés lors de l'appel. L'inférence reste possible tant qu'il n'y a pas d'ambigüité sur les types de donnée utilisés.

Comme nous l'avons vue, il est tout à fait possible de construire une classe et tous ses membres en utilisant des arguments de types.

```
TGenericType<AnyType> = class
  FData: AnyType;
  function GetData: AnyType;

  procedure UneMethodeGenerique<AnyType>(Variable: AnyType);
  function UneFonctionGenerique<AnyType>: AnyType;

  procedure SetData(Value: AnyType);
  property Data: AnyType read GetData write SetData;
end;
```

Cette construction génère l'avertissement suivant :

```
H2509 Identificateur 'AnyType' en conflit avec les paramètres type du type conteneur
```

car l'usage, dans la déclaration de la méthode **UneMethodeGenerique**, d'un paramètre type portant le même nom, **AnyType**, masque celui de la classe. On ne pourra donc pas accéder dans cette méthode au paramètre type de la classe. Si seul le type de l'argument nommé ***Variable*** doit suivre celui indiqué lors de la construction de la classe générique vous pouvez omettre le paramètre de type :

```
TGenericType<AnyType> = class
  FData: AnyType;
  function GetData: AnyType;

  procedure UneMethodeGenerique(Variable: AnyType);
  function UneFonctionGenerique: AnyType;

  procedure SetData(Value: AnyType);
  property Data: AnyType read GetData write SetData;
end;
```

Sinon utiliser un autre nom pour le paramètre de type :

```
  procedure UneMethodeGenerique<T>(Variable: T);
```

Notez que l'ajout des déclarations suivantes forcerait l'usage de la directive **Overload** sur la méthode *UneMethodeGenerique* :

```
  procedure UneMethodeGenerique(Variable: AnyType);Overlaod;
  procedure UneMethodeGenerique<R>(Variable: AnyType; Variable2: R);Overload;
  procedure UneMethodeGenerique<T,U>(Variable: U; Variable2: T);Overload;
```

6-6 - Les délégués

Comme les méthodes, les délégués permettent d'utiliser des gestionnaires d'événements génériques.

```
TOnMonEvenement<T> = procedure (Sender: TObject; var Valeur: T) of object;

TMaClasse<T>=class
 private
  FOnMonEvenement: TOnMonEvenement<T>;
 public
 property OnMonEvenement: TOnMonEvenement<T> read FOnMonEvenement write FOnMonEvenement;
end;
```

6-7 - Les interfaces

Projet : ..\Interfacegeneric

La définition d'interface permet l'usage de paramètre de type.

```
type
```

```
  IMonInterface<T>= interface
    procedure set_Valeur(const AValeur: T);
    function get_Valeur: T;
    property Valeur: T read get_Valeur write set_Valeur;
  end;

  IMonInterfaceDerivee<T>= interface(IMonInterface<T>)
   Procedure Multiplier(AMulplicateur:T);
  end;
```

A la compilation on verra que la procédure **Multiplier** ne peut être implémenté :

```
Procedure TClasseTest<T>.Multiplier(AMulplicateur:T);
begin
 //E2015 Opérateur non applicable à ce type d'opérande
 FCompteur:=FCompteur * AMulplicateur;
end;
```

Comme on ne connait rien du type T certaines opérations ne seront pas possible avec les génériques, comme ici l'utilisation de l'opérateur de multiplication.

6-8 - Pour information

Les énumérations ne peuvent pas être génériques. **Enumerations in the Common Type System** :

"*It is possible to declare a generic enumeration in Microsoft intermediate language (MSIL) assembly language, but a TypeLoadException occurs if an attempt is made to use the enumeration.*"

```
  Ensemble<T>=Set of T; //Erreur
```

Projet : ..\ClasseHelper

Les assistants de classe ne sont pas autorisés avec les types construits ouverts, le code suivant ne compile pas :

```
  TGenerique<T>  = class
   Champ: T
  end;

 THelperGenerique<T>=Class Helper for TGenerique<T>
 End;
```

```
 Erreur : E2508 Les paramètres de type ne sont pas autorisés sur ce type
```

En revanche il reste possible de les utiliser sur des types construits fermés :

```
  TGeneriqueInt=TGenerique<Integer>;
  THelperGeneriqueInt=Class Helper for TGeneriqueInt
   Procedure Test;
  End;
```

7 - Contraintes

Lors de la construction d'un type ou d'une méthode générique on peut contraindre un argument de type à respecter certaines règles.

Par exemple dans le cas où une classe générique utilise dans son code un itérateur **IEnumérable** on doit s'assurer que le type réel de l'argument de type rempli bien ce contrat. La présence de cette contrainte autorisera par là même la compilation du code utilisant cette interface.

Les mots clés **class**, **constructor** et **record** permettront d'en spécifier une ou plusieurs. Notez que la construction suivante ne stipule aucune contrainte :

```
TGenericType<T>
```


*Les mots clés **class** et **record** sont des contraintes exclusives.*

7-1 - Sur une classe ancêtre

On contraint l'argument de type **T** à dériver d'une classe ancêtre particulière, ici de la classe **TComponent** :

```
TGenericType<T:TComponent>
```

7-2 - Sur une ou plusieurs interfaces

On contraint l'argument de type à respecter le contrat d'une ou plusieurs interfaces génériques ou non.

L'exemple suivant contraint l'argument de type **T** à implémenter l'interface générique **IMonInterface<T>**

```
TGenericType<T:IMonInterface<T>>
```

7-3 - Sur l'accès à un constructeur sans paramètre

Projet : ..\ContrainteConstructeur

On utilisera le mot clé **constructor** pour contraindre un argument de type à posséder un constructeur sans paramètre et d'accès public.

On s'assure ainsi de pouvoir créer dans le corps des méthodes une instance du type passé en paramètre.

Voici un exemple :

```
  TGenericSansConstructeur<T> = class
  strict private
    FData: T;
    //Constructeur sans paramètre
    constructor Create; // inaccessible pour la contrainte :Constructor
  public
  ...
  end;

  TGenericAvecConstructeur<T> = class
  private
    FData: T;
  public
    //Constructeur sans paramètre
    constructor Create;
  ...
  end;

 TGenericContraint<ClasseInstanciable:Constructor>=Class
  private
   MaClasse : ClasseInstanciable;
  public
   Constructor Create;
 end;
```

Dans le code précédent la première classe possède un constructeur privé, la seconde un constructeur public et la troisième contraint l'argument de type. Notez que les arguments de type peuvent être des types 'classique', c'est à dire de types qui ne sont pas génériques.

Le code du constructeur, de notre classe générique **TGenericContraint**, nécessite de contraindre l'argument de type :

```
constructor TGenericContraint<ClasseInstanciable>.Create;
begin
 Inherited Create;
 MaClasse:=ClasseInstanciable.Create;
end;
```

Sans la contrainte **constructor** sur l'argument de type, on obtient, lors d'une tentative de création d'instance, l'erreur suivante :

```
 E2076 : Forme d'appel de méthode autorisée seulement pour méthodes de classe.
```

Le code suivant, de création d'une instance de **TGenericContraint**, compilera si la classe ou le record utilisé en paramètre possède un constructeur accessible et ne nécessitant aucun paramètre :

```
var  Z  : TGenericContraint<TGenericSansConstructeur<Integer>>;
     Z2 : TGenericContraint<TGenericAvecConstructeur<Integer>>;
begin
 try
  Z := TGenericContraint<TGenericSansConstructeur<Integer>>.Create;
  Z2:=TGenericContraint<TGenericAvecConstructeur<Integer>>.Create;
  WriteLn(Z2.MaClasse.ToString);
```

La création de l'objet **Z** avec le type **TGenericSansConstructeur** provoque l'erreur suivante :

```
E2513 : Le paramètre type 'ClassInstanciable' doit avoir un constructeur sans paramètre
```

Si le constructeur de la classe **TGenericAvecConstructeur** possédait au moins un paramètre, lors de l'exécution l'exception suivante est déclenchée :

```
Exception non gérée : System.TypeLoadException: GenericArguments[0], 'Contrainte
1.TGenericAvecConstructeur`1[System.Int32]', pour 'Contrainte1.TGenericContraint
`1[ClasseInstanciable]' ne respecte pas la contrainte du paramètre de type 'Clas
seInstanciable'....
```

7-4 - Sur un type référence

On utilisera le mot clés **class** pour contraindre l'usage d'un type référence, c'est à dire de n'importe quelle classe.

```
TGenericType<T:class>;
```

Ici **T** peut être de n'importe quelle classe ou plus précisément des classes qui dérivent de **System.Object**. Ce qui implique que les constructions suivantes sont autorisées :

```
Obj1 :TGenericType<TObject>;
Obj2 :TGenericType<IMonInterface>;
Obj3 :TGenericType<String>;
Obj4 :TGenericType<Array of Integer>;
...
```

7-5 - Sur un type valeur

On utilisera le mot clés **record** pour contraindre l'usage d'un type valeur, excepté les types Nullable.

```
TGenericClass<R:record>=Class //Contrainte record sur une classe, attend un paramètre de type d'un
type valeur.
 Champ: R;
End;
```

Ici **R** peut être de n'importe type valeur ou plus précisément des types dérivant de **System.ValueType**. Ce qui implique que les constructions suivantes sont autorisées :

```
Obj5: TGenericClass<TEnregistrement>;
Obj6: TGenericClass<Integer>;
Obj7: TGenericClass<Couleurs>; // avec  Couleurs=(Rouge,Noir,Vert);
```

7-6 - Sur un type nu (naked type)

Pour l'exemple suivant :

```
TClassContrainteTypeNu<S,U>=Class
  type
    TClassImbriquee<T:U>=Class
     Valeur:S;
```

```
      End;
  End;
```

" *L'argument de type fourni pour T doit être l'argument fourni pour U ou en dériver. C'est ce que l'on appelle une contrainte de type naked*", *dixit* MSDN.

La contrainte sur la classe imbriquée interdit, par exemple, la construction suivante :

```
var
 Parent:TClassContrainteTypeNu<String,Integer>;
 Naked : TObject;

begin
   Parent:=TClassContrainteTypeNu<String,Integer>.Create;
   Naked:=TClassContrainteTypeNu<String,Integer>.TClassImbriquee<String>.Create;
...
```

A la compilation la dernière ligne provoque l'erreur suivante :

```
E2515 Le paramètre type 'T' n'est pas compatible avec le type 'Integer'
```

On peut bien sûr coupler différentes contraintes en les séparant par une virgule et/ou les regrouper pour un ensemble d'argument de type :

```
TGenericType<T:Constructor,TMonItérateur,IEnumerable>=Class
 ...
end;

TGenericType<T,U:IEnumerable; S:record>=Class
 ...
end;
```

Projet : ..\ExemplesDeContraintes

8 - Héritage

Projet : ..\Heritage

Avec les génériques les possibilités d'héritage de classe se trouvent légèrement modifiées. Une classe de base peut hériter d'une classe générique et une classe générique peut hériter d'une autre classe générique ou d'une classe de base.

Voyons quelques exemples :

```
//Classe de base
 TClassDeBase=Class
  FData: integer;
 End;

  //Classe générique
 TGenerique<I>=Class
  FData: I;
 End;

  //Classe de base dérivée d'un type construit ouvert
 {TClassDeBaseDeriveDeGenerique=Class(TGenerique<I>) //E2003 Identificateur non déclaré : 'I'
  FData: integer;
 End;
 }
  //Classe de base dérivée d'un type construit fermé
 TClassDeBaseDeriveDeGenerique=Class(TGenerique<Integer>)
  FData: integer;
 End;

  //Classe générique dérivée d'une classe de base
 TGeneriqueDeriveeDeClass<I>=Class(TClassDeBase)
  FData: I;
 End;

  //Classe générique dérivée d'une classe générique
 TGeneriqueDeriveeDeGenerique<I,S>=Class(TGeneriqueDeriveeDeClass<I>)
  FData: I;
  Fitem: S;
 End;

  //Classe générique contrainte
 TGeneriqueContraint<I:constructor>=Class
  FData: I;
 End;

 //Classe générique dérivée d'une classe générique contrainte
  //Dans ce cas les contraintes doivent être redéclarées.
 TGeneriqueDeriveeDeGeneriqueContraint<I:constructor;S>=Class(TGeneriqueContraint<I>)
  FData: I;
  Fitem: S;
 End;
```

Vous remarquerez que les classes de base (non génériques) peuvent hériter des classes de base construites fermées, mais pas des classes construites ouvertes car dans ce cas on ne peut pas "propager" l'argument de type requis pour instancier la classe de base générique.

Notez également, dans le dernier exemple, que l'usage de classes génériques contraintes utilisées dans un héritage implique de redéclarer les mêmes contraintes dans la classe dérivée.

8-1 - Portée des paramètres de type

L'argument type T n'est visible que dans la déclaration et dans le corps des membres d'un type générique et pas dans les classes dérivées :

```
Type
   TParent<T> = class
     X: T;
   end;

   TEnfant<S> = class(TParent<S>)
     Y: T;  // Erreur! Identifieur inconnu "T"
   end;
```

9 - Surcharge de classe

Projet : ..\Surcharge

Nous avons pu voir rapidement dans le chapitre Comment ça fonctionne ?, que les génériques autorisent la surcharge de classe. Voyons les différents cas possibles :

```
TTest=Class
  FData: integer;
 End;
 //Surcharge générique
 TTest<I>=Class
  FData: I;
 End;

 {
 TTest<I:constructor>=Class //E2037 La déclaration de 'TTest<I>' diffère de la déclaration
précédente
  FData: I;
 End;
 }

 TTest<I,S>=Class
  FData: I;
  Fitem: S;
 End;

 //Surcharge générique à partir d'une autre surcharge générique de la même classe
     TGeneric<A,B>=Class
      FData: A;
      Fitem: B;
     End;

 TGeneric<A>=Class(TGeneric<A,String>)
  FData: A;
 End;

 //Surcharge générique à partir d'une autre surcharge générique d'une autre classe
 TGenericTest<R,U>=Class(TGeneric<R>)
  FData: R;
 End;
```

Notez que la surcharge basée sur une ou plusieurs contraintes est impossible.

Extrait des spécifications du C#:

> *Generic types can be "overloaded" on the number of type parameters; that is two type declarations within the same namespace or outer type declaration can use the same identifier as long as they have a different number of type parameters.*

9-1 - Surcharge d'opérateur

La **surcharge d'opérateur** reste possible avec les génériques :

```
TClasseOperateur<T> = class
   Champ1: T;
```

```
    Procedure Test(const Value: TOperateur<T>);
    class operator Negative(const Value: TOperateur<T>): TOperateur<T>;
end;
```

10 - Variable de classe dans les génériques

Extrait de la documentation de Delphi 2007 :

La variable classe définie dans un type générique est instanciée dans chaque type instancié identifié par les paramètres de type.
Le code suivant montre que ***TFoo<Integer>.FCount*** *et* ***TFoo<String>.FCount*** *ne sont instanciés qu'une seule fois, et que ce sont deux variables différentes.*

```
{$APPTYPE CONSOLE}
type
  TFoo<T> = class
    class var FCount: Integer;
    constructor Create;
  end;

constructor TFoo T>.Create;
begin
  inherited Create;
  Inc(FCount);
end;

procedure Test;
var FI: TFoo<Integer>;
begin
  FI := TFoo<Integer>.Create;
  FI.Free;
end;

var
  FI: TFoo<Integer>;
  FS: TFoo<String>;

begin
  FI := TFoo<Integer>.Create;
  FI.Free;
  FS := TFoo<String>.Create;
  FS.Free;
  Test;
  WriteLn(TFoo<Integer>.FCount); // renvoie 2
  WriteLn(TFoo<String>.FCount);  // renvoie 1
end;
```

11 - La classe System.Type et les génériques

Projet : ..\CreeGeneric

La classe **System.Type** du framework .NET 2.0 propose de nombreuses méthodes pour manipuler les types génériques.

Notamment la méthode **MakeGenericType** :

```
Procedure TConteneur.Main;
var t,
    generique,
    construit : System.Type;
    typeArgs    : Array of System.Type;

begin
 try
  WriteLn('Crée un type construit à partir du type générique Dictionary.',  Environment.NewLine);
   {Crée un type objet représentant le type générique Dictionary,
   en omettant le type arguments (mais en gardant la virgule qui les sépare,
   ainsi le compilateur peut déduire le nombre de paramètre de type).}
  generique:= typeof(Dictionary<,>);
  AfficheInformationDeType(generique);

   {Crée un tableau de type pour remplacer les paramètres de type de Dictionary.
   le paramètre 'Key' est de type String, et le type à contenir dans Dictionary est TConteneur}
  typeArgs:= TArrayType.Create(typeof(string), typeof(Self));

  {Crée un type objet représentant le type générique construit.}
  construit:= generique.MakeGenericType(typeArgs);
  AfficheInformationDeType(construit);

  {Compare les types d'objet obtenus ci-dessus aux types objet obtenus
  en utilisant les méthodes typeof et GetGenericTypeDefinition.}
  WriteLn(Environment.NewLine,'Compare les types obtenus par les différentes méthodes :');

  //t:= typeof(System.Collections.Generic.Dictionary<String, Self>);   //E2003 Identificateur non
 déclaré : 'Dictionary'
  t:= typeof(TGenDictionary);
  WriteLn('Les types construit sont-il égaux ? ', (t = construit));
  WriteLn('Les types génériques sont-il égaux ? ',(t.GetGenericTypeDefinition = generique));
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;
end;
```

Vous trouverez sur ce sujet un exemple plus concis dans le projet **CreeGeneric2**.

12 - L'instanciation des types génériques

Projet : ..\ConstructeurStaticGeneric

Extrait des **spécifications du C# 2.0 .**

*La première fois qu'une application crée une instance d'un type générique construit, tel qu'une pile, le compilateur just-in-time (JIT) du CLR de .NET convertis le code IL et les métas données du générique en code natif, substituant, dans le processus, les types réels aux types paramètres. Les références suivantes identiques à celle du type générique construit emploient le même code natif. Le processus de création d'un type construit spécifique à partir d'un type générique est connu sous le nom d'**instanciation de type générique**.*
Le CLR de .NET crée une copie spécialisée du code native pour chaque instanciation de type générique de type valeur, mais partage une simple copie du code natif pour tous les types références (puisque, au niveau de code natif, les références ne sont que des pointeurs avec la même représentation).

Ce qui peut être visualisé en partie par le code suivant au travers de l'appel au constructeur de classe :

```
type
 EContrainteArgumentException=Class(ArgumentException);

 MaClasse<T>=Class
   UnChamp : T;
   S       : String;
   Constructor Create;
   Class Constructor CreateClass;
  end;
...
var Objet : MaClasse<Integer> ;
    Objet2 : MaClasse<String>;
    Objet21 : MaClasse<TObject>;
    Objet3,Objet4 : MaClasse<Double>;
    Objet5 : MaClasse<Byte>;

begin
try
 Writeln('Début d''exécution du code.');
 Objet2:=nil;
  // Partage de code IL pour tous les types références
 Objet2:=MaClasse<String>.Create; //Appel du constructeur de classe, le type n'existe pas encore

 Objet21:=nil;
 Objet21:=MaClasse<TObject>.Create; //Appel du constructeur de classe, le type n'existe pas encore

  // Création de code pour chaque type valeur
 Objet3:=nil;
 Objet3:=MaClasse<Double>.Create; //Appel du constructeur de classe, le type n'existe pas encore

 Objet4:=nil;
 Objet4:=MaClasse<Double>.Create; //Pas d'appel du constructeur de classe, le type existe déjà

 Objet5:=nil;
 Objet5:=MaClasse<Byte>.Create; //Appel du constructeur de classe, le type n'existe pas encore
..
```

Nous pouvons, comme le suggère Patrick Smacchia dans son ouvrage **Pratique de .NET 2 et C# 2**, utiliser de façon particulière ces constructeurs de classe :

"*Si un type générique contient un constructeur statique, celui-ci est appelé à chaque création d'un de ses types génériques fermés. Nous pouvons exploiter cette propriété pour ajouter nos propres contraintes sur les types paramètres.*".

```
Class Constructor MaClasse<T>.CreateClass;
var UnEntier:Integer;
    VarGeneric :T;
begin
 Writeln(#9+#9+'Appel du constructeur de classe MaClasse.CreateClass<',Typeof(T),'>');
 VarGeneric:=default(T);

  //Est-ce un type valeur et le paramètre de type est-il un integer ?
 if (assigned(TObject(VarGeneric))=true) and (TObject(UnEntier) is T) then
  Raise EContrainteArgumentException.Create('L''utilisation du type Integer n''est pas autorisé
 pour la Classe MaClasse<T>');
end;
```

Notez que **Self** n'est pas accessible dans un constructeur de classe.

On ajoutera le bloc suivant pour la gestion des exceptions :

```
  //Appel du constructeur de classe, le paramètre de type Integer est interdit.
 Objet:=nil;
 Objet:=MaClasse<Integer>.Create;

 except
    //Déclenchée par le constructeur de la classe générique fermé MaClasse<Integer>
   on E:TypeInitializationException do
    If E.InnerException is EContrainteArgumentException
     then Writeln(E.InnerException.Classname, ': ', E.InnerException.Message)
     else Raise;
```

13 - Type Nullable

Projet : ..\TypeNullable

Le concept de **nullable** propose pour un type donné l'ajout d'un état supplémentaire à savoir l'état inconnu. Ainsi une variable de type **nullable** peut contenir toutes les valeurs possibles correspondant à son type, ainsi qu'une valeur additionnelle appelée *null*. La structure nullable prend en charge l'utilisation unique d'un type valeur comme type **nullable** parce que les types référence sont nullables de conception, et ce à l'aide du mot clé **nil**.

Ce qui est confirmé par le code C# du **code source CLI 2.0** :

```
public struct Nullable<T> where T : struct
```

13-1 - La déclaration

Le type **System.Nullable** étant un générique on déclare un type construit fermé :

```
var
    intNull : System.Nullable<integer>;
    dblNull : System.Nullable<Double>;
```

Si on souhaite déclarer une classe générique utilisant un argument de type nullable, on contraindra cet argument de type avec **record** :

```
TTestNullable<T:record>=Class
  procedure Inverser(Arg:System.Nullable<T>; Valeur:System.Nullable<T>);
End;
```

13-2 - L'assignation

L'assignation d'une variable de type null avec une valeur du type de base se fait simplement mais si on utilise une variable du type sous-jacent, ici **integer**, pour lui affecter la valeur d'un type null on doit le transtyper avec le type sous-jacent :

```
procedure Assignation;
var  I:Integer;
     intNull:System.Nullable<integer>;
begin
  intnull:=52;
  //I:=intNull;  //E2010 Types incompatibles : 'Integer' et 'Nullable<System.Int32>'
  I:=Integer(intNull); //Transtypage obligatoire
  I:=Convert.ToInt32(intNull);
```

L'affectation de **nil** ne fonctionnant pas pour signifier l'affectation d'une valeur null, on utilisera le mot-clé **Default** :

```
Writeln('Gestion d''une valeur null');
```

```
  //intnull:=Nil; //E2010 Types incompatibles : 'Nullable<System.Int32>' et 'Pointer'
  intnull:=Default(Nullable<integer>);
  I:=Integer(intnull);
```

La dernière ligne du code précédent déclenche l'exception **InvalidOperationException** car la variable *intNull* est à null. Dans ce cas afin d'éviter un bloc **try..except**, et si toutefois l'état null de la variable n'est pas déterminant dans le traitement, on utilisera la méthode **GetValueOrDefault** :

```
  I:=intNull.GetValueOrDefault;
```

13-3 - Opérateurs

Certaines opérations nécessiterons de manipuler la propriété en lecture seule nommée **Value**. Par exemple l'addition :

```
procedure Addition;
var  I       : Integer;
     J       : System.Nullable<integer>;
     intNull : System.Nullable<integer>;

begin
  IntNull:=1;
  //intNull:=intNull+1; //E2015 Opérateur non applicable à ce type d'opérande
  //intNull:=intNull+System.Nullable<integer>(1); //Idem
  //inc(intNull); //Idem
  //inc(intNull.value); //E2064 La partie gauche n'est pas affectable

  IntNull:=Integer(intNull)+1;
  intNull:=intNull.value+1;
  with intNull do
   intNull:=value+1;

  //I:=intNull+5; //E2015 Opérateur non applicable à ce type d'opérande
  I:=Integer(intNull)+5;
  I:=intNull.value+5;
  I:=7;
  J:=12;
  intnull:=J;
  intnull:=8;

 //Addition de deux variable de type null
  //intnull:=intNull+J; //E2015 Opérateur non applicable à ce type d'opérande
  //intnull:=intNull.Value+J; //idem
  intnull:=intNull.Value+J.Value;
  //with intNull do
   // Value:=Value+J.Value;   //E2129 Affectation impossible à une propriété en lecture seule
end;
```

Notez que l'opérateur **inc** ne semble pas implémenté, dans ce cas un assistant de classe sur un type nullable fermé autoriserait cette écriture.

13-4 - Diverses manipulations

Pour déterminer si une variable est à null ou pas, on utilisera soit directement la propriété **HasValue** :

```
procedure Test(Arg:System.Nullable<integer>);
begin
  if Arg.HasValue
```

```
    then writeln('La variable n''est pas à null')
    else writeln('La variable est à null')
end;
```

soit indirectement en accédant à la propriété **Value**, si cette variable est null alors l'exception **InvalidOperationException** est déclenchée :

```
procedure Test2(Arg:System.Nullable<integer>);
begin
 try
  Writeln('Valeur du type nullable =',Arg.Value);
  Except
    //l'accès à la propriété Value déclenche une exception si son contenu est Null
   On E:InvalidOperationException do
    Writeln('La variable est à null')
 end;
end;
```

Il est possible de déclarer un type nullable à partir d'un enregistrement :

```
 MonRecordInt=Record
   Data:Integer;
   Constructor Create (Value:Integer);
 End;
...
var RecNullable :System.Nullable<MonRecordInt>;
    Rec:MonRecordInt;
```

Dans ce cas on doit extraire le type sous-jacent pour le manipuler puis le réaffecter:

```
    RecNullable:=MonRecordInt.Create(852);
    Writeln(RecNullable.Value.Data);

    //RecNullable.Value:=10; //E2129 Affectation impossible à une propriété en lecture seule
    //RecNullable.Value.Data:=10; //E2064 La partie gauche n'est pas affectable
    Rec:=RecNullable.Value;
    Rec.Data:=10;

    RecNullable:=Rec;
    Writeln(RecNullable.Value.Data);
```

A moins de surcharger l'opérateur **implicit** :

```
<code langage="delphinet">
 MonRecordInt=Record
   Data:Integer;
   Constructor Create (Value:Integer);
   class operator Implicit(a: Integer): MonRecordInt;
 End;
class operator MonRecordInt.Implicit(a: Integer): MonRecordInt;
begin
 Result.Data:=a;
end;

...
    RecNullable:=MonRecordInt(99);  //Appel implicit
```

Comme notre enregistrement possède un seul champ cela reste possible.

14 - Les opérateurs is et as

L'opérateur **is** supporte les types ouverts.

Projet : ..\OperateurIS

Pour ce code :

```
type
 MaClasse<T>=class
  unchamp:T;
  procedure Test<X>;
  procedure Test2<X>;
 end;

 NullInteger = System.Nullable<Integer>;

procedure MaClasse<T>.Test<X>;
var Variable:X;
begin
 Writeln;
 Write(#9+'Le test sur X<',Typeof(X),'> is T<',typeof(T),'> est ' );
 if TObject(Variable) is T
  then writeln('Vrai')
  else writeln('Faux');
  if assigned(TObject(Variable))
   then writeln(#9+'assigned(Variable) =Vrai')
   else writeln(#9+'assigned(Variable)=Faux');
end;

procedure Test(Resultat:Boolean);
begin
 if Resultat
  then writeln('Vrai')
  else writeln('Faux');
end;

var Classe1: MaClasse<TObject>;
    Classe2: MaClasse<System.Nullable<Integer>>;
    Classe3: MaClasse<Integer>;
    VarNull: NullInteger;
    Objet  : TObject;
    int    : Integer;
```

le test (***TObject(Variable) is T***) du cas suivant :

```
 //If T is a nullable type, the result is true if D is the underlying type of T.
 Writeln('test pour la variable Classe2 MaClasse<System.Nullable<Integer>>');
 Classe2:=MaClasse<NullInteger>.Create;
 Classe2.Test<Double>;
 Classe2.Test<String>;
 Classe2.Test<integer>;
 Classe2.Test<NullInteger>;
```

donne ces résultats :

```
test pour la variable Classe2 MaClasse<System.Nullable<Integer>>

        Le test sur X<System.Double> is T<System.Nullable`1[System.Int32]> est Faux
        assigned(Variable) =Vrai

        Le test sur X<System.String> is T<System.Nullable`1[System.Int32]> est Faux
        assigned(Variable)=Faux

        Le test sur X<System.Int32> is T<System.Nullable`1[System.Int32]> est Vrai
        assigned(Variable) =Vrai

        Le test sur X<System.Nullable`1[System.Int32]> is T<System.Nullable`1[System.Int32]> est
 Faux
        assigned(Variable)=Faux
```

On peut voir que l'opérateur **is** se comporte différemment avec le type **Nullable**.

Le type sous-jacent du nullable, ici **System.Int32** est bien considéré comme un **System.Nullable<System.Int32>** mais l'inverse est faux.

Enfin, et à la différence du C#, le résultat de l'opérateur **is** entre une variable de type nullable et son type (**varNull is NullInteger**) renvoi toujours vrai que la variable soit ou non à *Null*.

*Le résultat de l'opérateur **is** avec des variables de type référence basées sur un argument de type (**Variable:X;**) sera toujours faux puisque sa valeur par défaut est égale à **nil**. Pour résoudre ce problème l'argument de type devra être contraint avec **constructor** afin d'autoriser, avant le test **is**, l'appel du constructeur. Et n'oubliez pas que les types valeur supportent aussi cette contrainte...*

L'opérateur **as** peut être employé avec un paramètre de type *T* en opérande droite.

Projet : ..\OperateurAS

```
Function MaClasse.Contraint<X>(AValue: TObject):X;
begin
 Result:=AValue as X;
end;
```

15 - Classes partielles

Les classes partielles ne sont pas possible sous Delphi .NET 2007.

16 - Net 3.0 et WPF

Le chargement de projet Delphi .NET 1.0 affichera le message d'information suivant :

```
Mise à niveau du projet ..\RAD Studio\Projets\Delegues\VCL\Delegues.bdsproj
Numéro de version de la personnalité "1.0" mis à jour vers "3.0"
```

Malheureusement les chemins des assemblies, pointant sur l'emplacement de l'ancienne version, sont à modifier manuellement.

Windows Communication Foundation (WCF) est le nouveau framework de transmission de messages de WinFX, la prochaine génération d'API Windows. WCF est responsable de la gestion des IHM sous Vista.

Conférence CodeGear sur **Windows Communication Foundation.**

Introduction to WCF Programming in Delphi

RAD Studio 2007 Developer Days replay, **WCF programming (.pdf 33 Mo)** by Pawel Glowacki.

Highlander: Convert HTML to Xaml Demo (.NET 3.0 / WPF).

Comparing WPF on Windows Vista v. Windows XP.

Package redistribuable de Microsoft .NET Framework 3.0 (nécessite une connexion internet pour continuer l'installation).

Télécharger le **SDK .NET 3.0** ou sa **mise à jour** de novembre 2006.

Introducing WPF: Chapter 1 of Essential Windows Presentation Foundation.

17 - Liens

MSDN: Génériques, **guide de programmation C#.**

C# .NET, les éléments principaux de la version 2.0.

Variance dans les types génériques.

Les spécifications du C# 2.0. Dans ce document ou prés de 40 pages sont dédiées aux génériques vous trouverez de nombreux exemples et points complémentaires qui n'ont pas été traités dans ce tutoriel.

Design and Implementation of Generics for the .NET Common Language Runtime.

Bjarne Stroustrup : le père de C++, un langage qui a de la classe

Différences entre les modèles C++ et les génériques C#.

Voir aussi l'ouvrage de Patrick Smacchia intitulé **Pratique de .NET 2 et C# 2** aux éditions O'Reilly.