Generics with Delphi 2009 Win32

With, as a bonus, anonymous routines and routine references

par Sébastien Doeraene (sjrd.developpez.com)

Date de publication : November 13th, 2008

Dernière mise à jour :

We have been waiting for them for a while! Now they are here: generics in Delphi Win32. Those little awesome things are coming with Delphi 2009. This tutorial will help you understand and use them, and design your own generic classes.

Your comments, critics, suggestions, etc. are welcomed on the blog.

This is a translation of a tutorial initially written in French: **Les génériques avec Delphi 2009 Win32**. I have translated my own tutorial, but I am no perfect English speeker. I therefore apologize for all the mistakes I certainly did in this translation.

Juan Badell has also translated this tutorial into Spanish: **Los genéricos en Delphi 2009** (**HTTP mirror**)

Aleksey Timohin has also made **a translation into Russian**.
This translation may be read in **PDF form**, which was built by **Andrew Tishkin** (**HTTP mirror**).

I - Introduction

I-A - Prerequisite

This tutorial was not written for the beginner in Delphi, or for a programmer who has never practised object oriented programming. It requires a good knowledge of the Delphi language, and non-negligible knowledge of OOP.

As for generics, no prior knowledge is *needed* for a good understanding of this tutorial. But, seeing there are already many existing articles on generics in other languages, I will not cover in details questions of matters like "How to use generics well?". The next section will introduce briefly the reader to the subject, but if you do not already know what generics are, the best understanding will come with the rest of the tutorial, with examples.

Here are some (French-speaking) articles covering the subject of generics in other languages/platforms. The concepts are always the same, and a fair amount of undertaking is as well.

- **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 - What are generics?

Generics are sometimes called *generic parameters*, a name which allows to introduce them somewhat better. Unlike a function parameter (argument), which has a *value*, a generic parameter is a *type*. And it parameterize a class, an interface, a record, or, less frequently, a method.

In order to ease the understanding immediately, here is an extract of the **TList<T>** class, which you may find in the **Generics.Collections.pas** unit.

Yes, it is possible to have dots in a unit name (except for that of the .pas, of course). This has nothing to do with the concept of namespace in .NET, neither the one of package in Java. It has simply the same meaning as a _ on that level: it is part of the name.

```
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;
```

You may readily see the weirdness of the type *T*. But is it really a type? No, it is a *generic parameter*. With this class at my disposal, if I need a list of Integers, I will use **TList<Integer>**, instead of a "simple" **TList**, along with many necessary casts in the code!

Generics thus allow, somehow, to declare a *set of (potential) classes* with a single code, or -more formally- a class *template*, but with one (or several) parameterized types, which you may change at will. Each time a new actual type is instanciated, you kind of write an entire new class, following its template, and, thus, make real one of those potential classes.

As I said before, I will not spread on the why and the how of generics. I will immediately go on with their daily use. If you think you still haven't got a clue on what I say, do not worry: everything is going to become clearer in a minute. You may also read the (French-speaking) tutorial **Les génériques sous Delphi .NET** written by Laurent Dardenne.

II - Daily usage: the example of TList<T>

Paradoxically, we will begin with the daily usage of a generic class -this is a misuse of language, one should say: generic class template-, instead of the design of such a type. There are several good reasons for this.

Firstly, it is considerably easier to write a class when you have a quite precise idea on how you are going to use it. This is more true yet when you discover a new programming paradigm.

Secondly, the majority of presentations of object oriented programming first explain how to use existing classes, as well.

II-A - A simple code to start with

Let us begin softly. We are going to write a small program listing the squares of integer numbers from 0 to X, X being defined by the user.

Without generics, we would have used a dynamic array (and it would have been far better, keep in mind this is a study case), but we are going to use an integer list.

Here is the 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;

var
  Max: Integer;
begin
  try
    WriteLn('Please enter a natural number:');
    ReadLn(Max);
    WriteSquares(Max);
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;

  ReadLn;
end.
```

What should one notice here?

First of all, of course, the declaration of the variable **List**, along with the creation of the instance. We have concatenated the actual parameter (or equivalent: it is a type, not a value) to the type name **TList**, between angle brackets < and >.

You can see that, in order to use a generic class, it is necessary, each time you write its name, to specify a type as a parameter. For now, we will always use a real type there.

Some languages allow *type inference*, i.e., the compiler guesses the type parameter. It is not the case in Delphi Win32. That is why you cannot write:

```
var
  List: TList<Integer>;
begin
  List := TList.Create; // <Integer> missing here
  ...
end;
```

The second thing is more important, yet lesser noticeable, since it is actually an absence. Indeed, no cast between an Integer and a Pointer is needed! Convenient, isn't it? And moreover, much more readable.

Another advantage is type safety: it is much better handled with generics. With casts, you always risk mistakes, e.g. adding a pointer to list intended for integers, or *vice versa*. If you correctly use generics, you will not probably need casts any more, or so few. Thereby, you will get less chances to make mistakes. Moreover, if you try and add a **Pointer** to an object of class **TList<Integer>**, the compiler will refuse your code. Before generics, such an error could have been revealed only by an absurd value, maybe months after releasing the software into production.

Note I took the type **Integer** on purpose, to limit code that is not directly related to the notion of generics, but one could have use *any type* instead.

II-B - Assignments between generic classes

As you know, when speaking of inheritance, we can assign an instance of a child class to a variable of a parent class, but not the other way. What about it with generics?

Start from the principle that you cannot do anything! Every following examples are *invalid*, and do not compile:

```
var
  NormalList: TList;
  IntList: TList<Integer>;
  ObjectList: TList<TObject>;
  ComponentList: TList<TComponent>;
begin
  ObjectList := IntList;
  ObjectList := NormalList;
  NormalList := ObjectList;
  ComponentList := ObjectList;
  ObjectList := ComponentList; // yes, even this is invalid
end;
```

As the comment implies, despite the fact that a **TComponent** may be assigned to a **TObject**, a **TList<TComponent>** may not be assigned to a **TList<TObject>**. In order to understand why, just think that **TList<TObject>.Add(Value: TObject)** would allow, if the assignment was valid, to insert a **TObject** value in a **TComponent** list!

The other important thing to notice is that **TList<T>** is *not* a specialization of **TList**, neither a generalization. Actually, they are *totally different types*, the former one being declared in the **Generics.Collections** unit, and the latter in **Classes**!

II-C - Methods of TList<T>

Interrestingly, **TList<T>** does not provide the same set of methods as **TList** does. We have more "high-level" methods at our disposal, but less "low-level" methods (as **Last**, which is missing).

The new methods are the following:

- 3 overloaded versions of **AddRange**, **InsertRange** and **DeleteRange**: they are equivalent to **Add/Insert/Delete** respectively, but for a list of elements;
- A **Contains** method;
- A **LastIndexOf** method;
- A **Reverse** method;
- 2 overloaded versions of **Sort** (whose name is not new, but whose usage is quite differente);
- 2 overloaded versions of **BinarySearch**.

I draw your attention to those changes, which you may find insignificant, because they are quite characteristic of the general changes in design concerns braught by generics.

What is the purpose, indeed, of implementing an **AddRange** method in the from now on obsolete **TList** class? None, since every item would have been cast, in turn. Therefore, one would have written a loop anyway, in order to build the array to insert. One might as well call **Add** in each loop iteration.

Meanwhile, with generics, the code can be written once and for all, and it becomes truly useful, for each and every type.

What you should remark and understand here, is that generics allow much more factoring of behaviors.

II-D - TList<T> and comparers

Certainly, **TList<T>** can handle any type. But how can it know how to compare two elements? How to know if they are equal, in order to search with **IndexOf**? How to know if one is lesser than another, in order to sort the list?

The answer is *comparers*. A comparer is an interface of type **IComparer<T>**. So yes, we are staying in the middle of generics. This type is declared in **Generics.Defaults.pas**.

When you instanciate a **TList<T>**, you may pass a comparer to the constructor, which will be used by all methods that need it. If you do not, a default comparer will be used.

The default comparer depends on the element type, of course. To get it, **TList<T>** calls the class method **TComparer<T>.Default**. This method does some nasty work, based on RTTI, to get the best possible solution. But does not always fit the requirements.

You may use the default comparer for the following data types:

- Ordinal types (integers, characters, booleans, enumerations);
- Floating point types;
- Set types (only for equality);
- Unicode long strings (**string**, UnicodeString and WideString);
- ANSI long strings (AnsiString), but *without page code* for < and >;
- Variant types (only for equality);
- Class types (only for equality - uses the **TObject.Equals** method);
- Pointer, meta-class and interface types (only for equality);
- Static and dynamic arrays whose elements are ordinal, floating point or set types (only for equality).

For all other types, the default comparer compares only the memory contents of the variable. One should therefore write a custom comparer.

There exist two simple ways. The first one is based on writing a function, the other one on the derivation of the class **TComparer<T>**. We will illustrate both of them by implementing comparison for **TPoint**. We will assume that points are compared according to their distance to the origin -the point (0, 0)- in order to have a total ordering (in the mathematical meaning).

II-D-1 - Writing a comparer using TComparer<T>

Nothing easier, you have always done that! An only method to override: **Compare**. It must return 0 on equality, a positive integer if the first parameter is greater than the second one, and a negative integer otherwise.

Here is the result:

```
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;
```

*Notice the parent class of **TPointComparer**: it inherits from **TComparer<TPoint>**. You can see that it is possible for a "simple" class to inherit from generic class, provided it is given an actual parameter for its generic parameter.*

In order to use our comparer, just instantiate it and pass the instance to the list constructor. Here is a small program that creates 10 random points, sort them and prints the sorted list.

```
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; // uses the comparer given to the constructor

    for Item in List do
```

```
      WriteLn(Format('%d'#9'%d'#9'(distance to origin = %.2f)',
        [Item.X, Item.Y, DistanceToCenter(Item)]));
  finally
    List.Free;
  end;
end;

begin
  try
    Randomize;

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

  ReadLn;
end.
```

*Wait a minute! Where is the comparer being freed? Recall that **TList<T>** takes an interface of type **IComparer<T>**, not a class. Because of reference counting on interfaces (implemented correctly in **TComparer<T>**), the comparer will be automatically freed when it is no longer needed.*
*If you do not know anything about interfaces in Delphi, I recommand you read the (French-speaking) tutorial **Les interfaces d'objet sous Delphi** written by Laurent Dardenne.*

II-D-2 - Writing a comparer using a simple comparison function

This alternative seems to be simpler, given its name: no need to play with additional classes. However, I have chosen to present it second, because it introduces a new data type available in Delphi 2009. I am speaking of routine *references*.

Er ... I know routine references! Hum, well, no, you do not ;-) What you already know are *procedural types*, for example **TNotifyEvent**:

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

Routine reference types are declared, in this example, like **TComparison<T>**:

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

There are at least three differences between procedural types and routine reference types.

Firstly, a routine reference type cannot be marked as **of object**. In other words, you can never assign a method to a routine reference, only... routines. (Or, at least, I have not been successful in trying to do so ^^.)

The second difference is more fundamental: while a procedural type (non **of object**) is a pointer to the base address of a routine (its entry point), a routine reference type is actually an *interface*! With reference counting and this kind of things. However, you probably will never have to care about that, because its daily usage is identical to that of a procedural type.

Lastly -and this explains the apparition of routine references-, one can assign a *anonymous routine* (we will see in a moment what it is like)- to a routine reference, but not to a procedural type. Try to do so, you will soon see that the compiler reports errors. Incidentally, that explains also why routine references are implemented as interfaces, but thoughts on this subject are not within the framework of this tutorial.

Let us get back to our point sorting. In order to create a comparer on the basis of a function, we use another class method of **TComparer<T>**; it is **Construct**. This class method takes as parameter a routine reference of type **TComparison<T>**. As was already said, using routine references is quite similar to the use of procedural types: one may use the routine name as parameter, directly. Here is the code:

```
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; // uses the comparer given to the constructor

    for Item in List do
      WriteLn(Format('%d'#9'%d'#9'(distance to origin = %.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;

  ReadLn;
end.
```

The only difference coming, of course, from the creation of the comparer. The rest of the usage of the list is strictly identical (just as well!).

Internally, the class method **Construct** creates an instance of **TDelegatedComparer<T>**, which takes as parameter of its constructor the routine reference which is going to handle the comparison. Calling **Construct** thus returns an object of this type, encapsulated in an interface **IComparer<T>**.

Well, it was finally easier as well. Actually, on should realize it: generics are there to ease our life!

But I have let something go: one can assign a anonymous routine to a routine reference. So, let us see what would it be like:

```
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));

  // Always the same ...
end;
```

This kind of create is interresting mostly if it is the only place where you need the comparison routine.

Speaking of anonymous routines: yes, they may access local variables of the enclosing routine/method. And yes, they may do so even after the enclosing routine/method has returned. The following example illustrates it all:

```
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;

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

  // Always the same ...
end;
```

Interresting, isn't it?

Voilà! Here is the end of our small tour of comparers used with **TList<T>**, and, with it, this introduction to generics through the use of this class. In the following chapter, we will begin to learn how one can write his own generic class.

III - Design of a generic class

Now that we know how one can use generic classes, it is time to see how does a generic class work internally.

Therefore, we are going to develop a **TTreeNode<T>** class, which is going to be a generic implementation of a tree. This time, it will not be a study case any more. It is a real class, which you will be able to use in your real projects.

III-A - Class declaration

Let us begin with the beginning: the class declaration. As you may recall from the extract of the declaration of **TList<T>**, the generic parameter (traditionally called **T** when it has no precise meaning) is added between angle brackets. On should then write:

```
unit Generics.Trees;

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

A tree node will be labelled by a value of type **T**, and will have 0 to many children. When a node is destroyed, it frees all its children.

A value of type **T**? Well, this is as simple as this: **FValue: T;** the same way as for any normal type. To keep a list of the children, we will use a **TObjectList<U>**, declared in **Generics.Collections**. I have used **U** on purpose here, because I do not want you to be misleaded. Actually, **U** will be set to **TTreeNode<T>**! So yes, one can use a generic class as an actual generic parameter.

*We do not implement a search tree. Consequently, we do not need any comparer like **TList<T>** does.*

Our class will then have the following fields:

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

Weird? If we think a bit about it, not that much. We have said previously that a generic type could be replaced by *any* type. So, why not a generic class type?

*I draw your attention to the fact that, in the name **TTreeNode<T>**, **T** is a non actual generic parameter (a formal paremeter, if you prefer). But in the inside of the class, it becomes a actual type! It behaves exactly as with the parameters of a routine. In the signature, you have formal parameters; but in the routine body, they are local variables just as much as any other. That is why on can use **T** as a type for **FValue**, or even as an actual parameter parameterizing **TTreeNode<T>**, in the declaration of **FChildren**.*

*When I said that we could use any type as actual parameter, I sort of lied to you. It is not exactly always true. For example, in **TObjectList<T>**, **T** must be a class type. This is so because **TObjectList<T>** has imposed a constraint on its parameterized type. We will see*

*this concept later in more details, but I must point it out now, since we are using this class. We use it because we want to take profit from the automatic freeing of the objects stored in a **TObjectList<T>**, **TList<T>** not doing it.*

As methods, besides the constructor and destructor, we provide methods for in-depth walks (preorder and postorder), along with methods for adding/moving/deleting child nodes. Actually, the addition will be asked by the child itself, when it will be given its parent.

For the walks, we will need a call-back type. Guess what? We will use a routine reference type. And we will provide two overloads for each walk: a walk on the nodes, and a walk on the values of the nodes. The second one will probably be used more often when using the **TTreeNode<T>** class. The first one is provided for completeness, and is more general. It will be used by several methods of **TTreeNode<T>** (e.g. the second overload).

Finally, there will be, of course, properties accessing the parent, children and labelled value. Which leads to the code below. You may see there is nothing new, besides the fact there are many **T**'s everywhere.

Here is the complete declaration of the class -which, be reassured, I give you after I have completely impleted the class ^^.

```
type
  /// Call-back routine reference with an only parameter
  TValueCallBack<T> = reference to procedure(const Value: T);

  {*
    Generic tree structure
  *}
  TTreeNode<T> = class(TObject)
  private
    FParent: TTreeNode<T>;                 /// Parent node (nil ifor the root)
    FChildren: TObjectList<TTreeNode<T>>; /// List of the children
    FValue: T;                             /// Label

    FDestroying: Boolean; /// True when the object is being destroyed

    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;
```

Let us identify what is remarkable here. Actually, not much, beside the fact that every parameter of type **T** is declared as **const**. Indeed, we do not know, at the time of writing, what could be the actual **T**. And thus there are some possibilities for it being a "heavy" type (as a string or record), for which it is more efficient to use **const**. And, as **const** is never penalizing (it is effectless on "light" types, such as Integer), we always put it when working with generic types.

*The concept of heavy and light types is a concept of my own, nothing official, hence the quotes. I intend for heavy type a type whose parameters are better (in execution time) passed as **const** when it is possible. Those types are, in a general view, those who stretch on more than 4 bytes (floats and Int64 excluded), and those who require an initialization. One should remember strings, records, arrays, interfaces (not classes) and Variants.*

However, when a parameter is of type **TTreeNode<T>**, we do not put **const**. Indeed, no matter what is **T**, **TTreeNode<T>** will remain a class type, which is a light type.

III-B - Implementing a method

In order to illustrate the particularities of the implementation of generic method, let us work with **GetRootNode**, which is very easy. It is written as below:

```
{*
  Root node of the tree
  @return Root node of the tree
*}
function TTreeNode<T>.GetRootNode: TTreeNode<T>;
begin
  if IsRoot then
    Result := Self
  else
    Result := Parent.RootNode;
end;
```

The only thing to notice here is that, *each* time your write the implementation of a generic method, you must specify the <T> next to the class name. It is mandatory, indeed, because **TTreeNode** (without <T>) would be another type, maybe existing in the same unit.

III-C - The pseudo-routine Default

In order to implement the two constructors that have no **AValue** parameter, we will need to initialize **FValue** one way or the other. Sure but, since we do not know the actual type of the value, how can we write a default value, for any possible type?

The solution is the new pseudo-routine **Default**. As well as **TypeInfo**, this pseudo-routine takes, as parameter, a type identifier. And the name of a generic type is actually a type identifier.

This pseudo-routine "returns" the default value of the provided type. One can then write:

```
{*
  Create a node with a parent and without label
  @param AParent   Parent
*}
constructor TTreeNode<T>.Create(AParent: TTreeNode<T>);
begin
  Create(AParent, Default(T));
end;

{*
  Create a node without parent and without label
*}
constructor TTreeNode<T>.Create;
begin
  Create(nil, Default(T));
end;
```

In the case of initializing an object field, this is not, strictly speaking, necessary, because instanciation of an object already initializes all its fields to their respective default value. But I could not find a better place to introduce you with this pseudo-routine.

III-D - Routine references with tree walks

In order to illustrate routine references, from the other side of the mirror, let us discuss the implementation of tree walks.

Both overloads of the walk takes as parameter a call-back routine reference. Notice that, also here, we have used a **const** parameter. Indeed, the attentive reader will remeber that routine references are actually interfaces. Now, an interface is a heavy type. One should then use **const** when possible.

Except this little consideration, there is nothing special to say on the first version, the one on nodes:

```
{*
  Preorder walk on the nodes of the tree
  @param Action   Action to execute on each node
*}
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;
```

To call the call-back, we have written exactly the same piece of code as with procedural types, i.e. the same form as a routine call, but with the routine reference variable instead of the routine name.

When implementing the second overload, we use the first one, giving for **Action**... a anonymous routine!

```
{*
  Preorder walk on the values of the tree
  @param Action   Action to execute on each value
*}
procedure TTreeNode<T>.PreOrderWalk(const Action: TValueCallBack<T>);
begin
  PreOrderWalk(
    procedure(const Node: TTreeNode<T>)
    begin
      Action(Node.Value);
    end);
end;
```

Isn't it a nice piece of code?

*You will say I ramble on **const** parameters, yet: the **Node** parameter of the anonymous routine has been declared **const**, yet it is clearly a class type. The reason is: one should nertheless be compatible with the signature of the **TValueCallBack<TTreeNode<T>>** type, which demands a **const** parameter ;-).*

III-D-1 - Other methods using anonymous routines?

Yes, two others. **DoAncestorChanged**, which has to undertake a preorder walk on the method **AncestorChanged**. And **BeforeDestruction**, which has to do a preorder walk on the **Destroying** method of all the children. It is always the same:

```
{*
  Call the AncestorChanged procedure on every descendant (preorder)
*}
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 - And the rest

The rest, well ... That is pretty classic ;-) Nothing new. I will not extend on this, but the complete source is available for download, as all the others, **at the end of the tutorial**.

III-F - How to use TTreeNode<T> ?

That is certainly nice to write a generic tree class, but can we use it?

After a description in every detail of the **TList<T>** class, there is not much to say left. I am only going to give you the code of a small test program.

```
procedure TestGenericTree;
const
  NodeCount = 20;
var
  Tree, Node: TTreeNode<Integer>;
  I, J, MaxDepth, Depth: Integer;
begin
  Tree := TTreeNode<Integer>.Create(0);
  try
    // Build the tree
    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;

    // Show the tree with a preorder walk
    Tree.PreOrderWalk(
      procedure(const Node: TTreeNode<Integer>)
      begin
        Write(StringOfChar(' ', 2*Node.Depth));
        Write('- ');
        WriteLn(Node.Value);
      end);
  finally
    Tree.Free;
  end;
end;
```

IV - Design of a generic record

Let us have a look at something else, and let us develop the simple yet useful record **TNullable<T>**. Its goal is to have either a value of type **T**, either **nil**. It is highly likely that you have already needed such a type, for example to represent the NULL value of databases.

This record will contain two fields: **FValue** of type **T**, and **FIsNil** of type **Boolean**. Two properties are provided for read access to those fields. We will use only implicit convertion operators to build values of this 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;
```

For more information about operator overloading, I recommand you read the (French-speaking) tutorial ***La surcharge d'opérateurs sous Delphi 2006 Win32****, written by Laurent Dardenne.*

So, it is a *immutable* type (whose state cannot be changed once created).

The implementation of three convertion operators is quite easy. The second one (the one with a **Pointer** parameter) is provided to allow the assignment := **nil**.

```
uses
  SysUtils;

resourcestring
  sCantConvertNil = 'Cannot convert to nil';
  sOnlyValidValueIsNil = 'The only valid value is 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;
```

This small record can be used with this kind of code:

```
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;
```

The exection of which yields:

```
5
nil
10
```

Understood it all? That is great, because I have not given a single piece of explanation. Proof is given that generics are as easy as pie :-).

The complete source code of **Generics.Nullable** is available for download **at the end of the tutorial**.

V - Contraints on generic types

OK, you now know the basis. It is time to move on to serious things, namely, constraints.

As the name implies, constraints allow to impose some restrictions on the actual types that can be used to parameterize a generic type. Continuing the comparison with method parameters: a constraint is to the type what a type is to a variable. Not clear? OK, when you specify a type for a parameter, you may only pass it an expression compatible with this type. When you specify a constraint on a type parameter, you must replace it by an actual type satisfying this constraint.

V-A - What are the available constraints?

There are few available constraints. Actually, there are four kinds of constraints.

On may force a generic type to be:

- a class type, descendant from a given class;
- an interface type, descendant from a given interface, or a class type implementing this interface;
- an ordinal, floating point or record type;
- a class type that provides a zero-argument constructor.

In order to impose a constraint to a generic parameter, one should write:

```
type
  TStreamGenericType<T: TStream> = class
  end;

  TIntfListGenericType<T: IInterfaceList> = class
  end;

  TSimpleTypeGenericType<T: record> = class
  end;

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

Those syntaxes impose, respectively, that **T** must be replaced by the **TStream** class or one of its descendants; by **IInterfaceList** or one of its descendants; by an ordinal, floating point or record type (a non-null data type, following the terminology of Delphi); or finally by a class type providing a zero-argument constructor.

*<T: class> must be used instead of <T: TObject> ... We understand that **class** be accepted, but we may ask why **TObject** is rejected.*

It is possible to combine several interface constraints, or a class constraint along with one or several interface constraints. In this case, the actual type must satisfy *all* the constraints at the same time. The **constructor** constraint may also be combined with class and/or interface constraints. It is even possible to combine the record constraint with one or several interface constraint, yet I cannot see an actual type that could statisfy the two constraints at once (in .NET, it is possible, but, for now, not in Win32)!

*Maybe -but this is pure speculation of mine- this is in anticipation of a future version, where the Integer type, for example, would "implement" the **IComparable<Integer>** interface. It would then statisfy both constraints of a declaration like **<T: record, IComparable<T>>**.*

It is also possible to use a generic class or interface as constraint, with a specified type parameter. This parameter could by **T** itself. For example, one might demand a type which can compare to itself. One would write:

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

Additionnaly, if you want **T** to be a class type, you may combine:

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

V-B - But what is the purpose of it?

"I thought generic were designed to write a code once and for all for *all* types. Then, what is the interest of *limit* the possible types?"

Well, it allows the compiler to get more knowledge about the type used. For example, it allows it to know, with a constraint <T: class>, that it is legal to call the method **Free** to a variable of type **T**. Or, with a constraint <T: IComparable<T>>, that one may write **Left.CompareTo(Right)**.

We are going to illustrate that with a child class of **TTreeNode<T>**, called **TObjectTreeNode<T: class>**. Following the example of **TObjectList<T: class>**, which provides an automatic freeing of its elements when destroyed, our class will release its labelled value on destruction.

Actually, there is not much code, considering we have already written most of it in the superclass.

```
type
  {*
    Generic tree structure whose values are objects
    When the node is destroyed, its labelled value is freed as well.
  *}
  TObjectTreeNode<T: class> = class(TTreeNode<T>)
  public
    destructor Destroy; override;
  end;

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

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

That is all. The aim of this example is only to show the technique. Not to have an exceptionnal code.

You should notice two things here. First, a generic class may inherit from another generic class, using again the same generic parameter (or not).

Then, in the implementation of methods from generic classes with constraints, the constraints must not be repeated.

You may, by the way, try and delete the constraint and compile. The compiler will raise an error on the call to **Free**. Indeed, **Free** is not available on all types. But it is on all *class* types.

V-C - A variant with constructor

You might as well want the two constructors without the **AValue** parameter to create an object for **FValue** instead of using **Default(T)** (the latter returning **nil** here because **T** is constrained to be a class).

The **constructor** constrained is intended for this purpose, which yields:

```
type
  {*
    Generic tree structure whose values are objects
    When a node is created without labelled value, a new value is created with
    the default (zero-argument) constructor of T.
    When the node is destroyed, the labelled value is freed as well.
  *}
  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;
```

Again, if you remove the **constructor** constraint, you will get a compiler error on **T.Create**.

VI - Parameterizing with more than on type

As you might have already thought, it is possible to parameterize a generic type with several types. Each of them having its own set of constraints.

In this way, the **TDictionary<TKey,TValue>** class takes two type parameters. The first one being the type of the keys, and the second one being the type of the elements. This class implements a hash table.

*do not be mistaken: **TKey** and **TValue** are indeed (formal) generic parameters, not real types. do not mix up things because of the notation.*

The declaration syntax is quite lax, on this point. It is indeed possible to separate the types with commas (,) or semi-colons (;). Even mixing is accepted when there are more than two types. As much at the declaration level as at the implementation level. However, when *using* a generic type, you must use commas!

This said, if you place one or more constraints on a type which is not the last one, you will have to use a semi-colon to separate it from the next one. Indeed, the comma would mean an additional constraint.

So, let me suggest you a style rule -which is not the one followed by Embarcadero. Always use the semi-colons in the declaration of generic types (where it is liable to have a constraint) and use commas everywhere else (implementation of methods, and usage of generic types).

As I have no better example of generic type with two parameters to offer you than **TDictionary<TKey,TValue>**, I suggest you have a look at the code of this class (defined, as you might have guessed, in **Generics.Collections**). Here is an extract of it:

```
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;
```

*As you already noticed, **TDictionary<TKey,TValue>** uses generic type names that are more explicit than the ever-lasting **T** we have used so far. You should do the same, whenever the type has a particular meaning, which is the case here. All the more when there are more than one parameterized type.*

VII - Other generic types

Until now, we have only defined generic *classes* and *records*. Yet, we have already crossed over some generic *interfaces*.

It is also possible to declare generic *array* types (static or dynamic), where only the *elements* type may depend on the generic type (not the dimensions). But it is quite unlikely that you will find a practical situation for them.

So, it is not possible to declare a generic pointer type, nor a generic set type:

```
type
  TGenericPointer<T> = ^T; // compiler error
  TGenericSet<T> = set of T; // compiler error
```

VIII - Generic methods

We have explored so far the many possibilities offered by generics, applied to types defined by the developer. It is time to move on to generic *methods*.

In many presentations of generics and templates in other languages, this use of generics is explained first. Again, I have preferred to begin with generic *types*, which are used more often.

VIII-A - A generic Min function

In order to introduce the concept, we will write a class method **TArrayUtils.Min<T>**, which finds and returns the smallest element of an array. We will then, of course, need a comparer of type **IComparer<T>**.

Like a generic type name, a generic method name must be followed by the generic parameters, between angle brackets. Here, the generic type is the type of the elements of the array.

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

No, it is not possible to declare a global routine with generic parameters. A possible reason for this limitation is due to a parallelism with the Delphi.NET syntax, in order to reduce development and maintenance efforts, internally.

*To make up for this limitation, we then use class methods. Better declared as **static**. This has nothing to do with the homonym keyword in C++. It has to do here with static linking. This is to say, in such a method, there is no **Self**, and then calls to virtual class methods, or to virtual constructors, are to "virtualised". In other words, it is like they were not virtual any more.*
Finally, it results in a genuine global routine, but with a different namespace.

*Unlike some other languages, it is not necessary for a parameter at least to use the generic type. Thus, it is possible to write a method Dummy<T>(Int: Integer): Integer, which has no formal paremeter whose type depends on the generic parameter **T**. In C++, for example, that would not get pass compilation.*

The implementation of the method is quite similar to classes. You have to repeat the angle brackets with the generic types, but not their possible constraints. Our Min method will then look like this:

```
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;
```

Nothing exceptional, as you may see ;-)

VIII-B - Overloading and constraints

Let us take advantage of this nice example to review our constraints, and provide an overloaded version for those element types which support the **IComparable<T>** interface (this interface is declared in **System.pas**).

Before that, let us add another overload which takes a routine reference of type **TComparison<T>**. Remember we can easily "convert" a **TComparison<T>** call-back to a **IComparer<T>** interface with the **TComparer<T>.Construct** method.

You may observe the usage of the method **CompareTo** on the **Left** parameter. This call is of course permitted only because of the constraint.

```
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;
```

*Have a look at the call to **Min<T>**: specifying the actual type between angle brackets is mandatory even there. This is not the case in other languages like C++.*

Now, we want to provide a fourth overloaded method, again only with the **Items** parameter, but with an unconstrained **T** parameter. This version would use **TComparer<T>.Default**.

But this is impossible! Indeed, even though the constraints on types do change, the *parameters* (arguments) are still the same. So, the two overloads are totally ambiguous! Therefore, the following declaration would result in a compiler error:

```
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; // Compiler error
  end;
```

Then, we must choice: abandon either the first or the second, or else use another name. And as, until base types like **Integer** support the **IComparable<T>** interface, you will need both as often, you will have to opt for another name ;-)

```
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;
```

Why did I cast explicitly **Default** into an **IComparer<T>**, though it is obviously *already* an **IComparer<T>**? Because routine references and overloads still do not work very well with each other, and the compiler seems to get confused with the mixing. Without the cast, the compilation fails ...

VIII-C - Some extras for TList<T>

Even though the **TList<T>** is quite a good innovation, it could nevertheless provide some more practical methods.

Here is, for example, an implementation of the .NET **FindAll** method for **TList<T>**. This method aims to select a sub-list from a predicate function. What we call a predicate function is a call-back routine that takes an element of the list, and returns **True** if it should be selected, **False** otherwise. We define a routine reference type **TPredicate<T>** as follows:

```
unit Generics.CollectionsEx;

interface

uses
  Generics.Collections;

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

Then, since it is unfortunately impossible to write a **class helper** for a generic class, we will write a class method **FindAll<T>** which is going to to that. Since we are deprived of **class helper**, we will at least take advantage of it to be more general, working with an enumerator, or an enumerable.

```
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.
```

One can use this method as follows:

```
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;
```

Again, the cast is necessary because of overloads. This is quite deplorable, but yet it is so. If you prefer not to have casts, use different names, or get rid of one of the two overloads.

It is up to you to complement this class with other methods like **FindAll** :-)

IX - RTTI and generics

In this last chapter, I am going to give some information about what generics changed to RTTI. If you never play with RTTI, you should skip this entire chapter. It is not intended at all to be an introduction to RTTI.

IX-A - Changes to the pseudo-routine TypeInfo

RTTI always begin with the pseudo-routine **TypeInfo**. You may know that certain types cannot be applied to this pseudo-routine; for example, pointer types. And that, because of that, it never returns **nil**.

So, may we call **TypeInfo** on a generic type **T**? The question is pertinent: **T** might as well be a pointer type (invalid for **TypeInfo**) or an integer type (valid).

The answer is *yes*, one can call **TypeInfo** on a generic type. But what happens then if **T** results in a type that has no RTTI? Well, in this only case, **TypeInfo** returns **nil**.

To illustrate the phenomenon, here is a small class method that prints the name and the kind of type, but *via* generics:

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

class procedure TRTTI.PrintType<T>;
var
  Info: PTypeInfo;
begin
  Info := TypeInfo(T); // warning ! Info may be nil here

  if Info = nil then
    WriteLn('This type has no RTTI')
  else
    WriteLn(Info.Name, #9, GetEnumName(TypeInfo(TTypeKind), Byte(Info.Kind)));
end;
```

This method is quite special, in the sense that its real parameter is actually passed as a parameterized type.

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

Whose execution yields:

```
Integer  tkInteger
TObject  tkClass
This type has no RTTI
```

IX-A-1 - A more general TypeInfo function

I have already regretted that **TypeInfo** could not be applied to any type, even for a **nil** reply; maybe you did as well. Then, here is a small replacement method solving the method.

```
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;
```

Which you may use as:

```
Info := TRTTI.TypeInfo<Pointer>; // Info = nil
// instead of :
Info := TypeInfo(Pointer); // compiler error, since Pointer has no RTTI
```

IX-B - Do generic types have RTTI?

There are two questions here: do *non instaciated* generic types (with a non defined **T** parameter) have RTTI? And do *instanciated* generic types (where **T** has been replaced with an actual type, like **Integer**) have RTTI?

The easiest way to know it is to try ;-) You may observe, with a simple test, that only *instanciated* generic types. Actually, that is quite logical, since non instanciated generic types are not actual types, but type *templates*, and do not exist once compilation is done.

One can ask himself what *name* would such a type have. So:

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

Which yields:

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

This shows that the name includes, between angle brackets, the fully qualified name of the actual type replacing the type parameter.

*You can also notice that the routine reference type **TComparison<T>** yields **tkInterface**, which proves that it is actually an interface.*

That is all I wanted to say about RTTI and generics. I will not of course speak about other changes brought to RTTI in Delphi 2009 because of Unicode strings.

X - Conclusion

This is the end of our trip among generics with Delphi 2009. If you have not already had the chance to experiment generics in other languages, that might appear disturbing. But they are actually very practical, and do change the developer's life, as soon as he plays a bit with them.

XI - Download the sources

All the sources of this tutorial (documented in French) may be found in the zip file Source **sources.zip** (Source **mirroir HTTP**).

XII - Acknowlegments

I would like to thank **Laurent Dardenne** and **SpiceGuid** for their many comments and corrections, which have allowed this tutorial to be greatly improved.

Thanks go to **Thibaut Cuvelier** as well, for his corrections about this English translation.