Los genéricos en Delphi 2009

Con propina sobre rutinas anónimas y las referencias de rutina

por Sébastien Doeraene (sjrd.developpez.com)
traducción por Juan Badell
Fecha de Publicación: 13 noviembre 2008
Última puesta al día:

¡ Hace tiempo que los esperábamos ! Finalmente están aquí: los genéricos en Delphi Win32. Estas pequeñas maravillas llegan con Delphi 2009. Este tutorial te da opción a comprenderlos, aprender a utilizarlos y, a continuación, diseñar tus propias clases genéricas.

Vuestros comentarios, críticas, sugerencias, etc. serán bienvenidos en el blog.

I. Introducción

I-A. Requisitos

Este tutorial no es para el principiante en Delphi que nunca ha practicado la programación orientada a objetos. Requiere conocer el lenguaje de Delphi antes, y un importante conocimiento de la POO.
En cuanto a los genéricos, no es necesario el conocimiento previo para la correcta comprensión y asimilación de este tutorial. Sin embargo, dado el número de artículos sobre los genéricos existentes en otros idiomas, no cubriré en detalles las preguntas como "¿Cómo usar los genéricos correctamente?". La siguiente sección introduce rápidamente al lector en el concepto de los genéricos, pero si usted no sabe aún lo que es, la mejor comprensión se hará siguiendo este tutorial a través de ejemplos. Aquí hay algunos artículos que tratan los genéricos en otros lenguajes / plataformas. Los conceptos son siempre los mismo, y gran parte de su puesta en escena también.

- Les génériques sous Delphi .NET por Laurent Dardenne ;
- Generics avec Java Tiger 1.5.0 por Lionel Roux ;
- Cours de C/C++ - Les template por Christian Casteyde.

I-B. ¿Qué son los genéricos?

Los genéricos, a veces llamados parámetros genéricos, un nombre que permite presentarlos mucho mejor. A diferencia del parámetro de una función (argumento), que tiene un valor, el parámetro genérico es un tipo. Y un parámetro genérico parametrizado una clase, un interfaz, un record, o, con menor frecuencia, un método.

Para dar inmediatamente una idea más clara, este es un extracto de la clase **TList<T>**, usted puede encontrarlo en la unidad **Generics.Collections.pas**.

¡Sí!, es posible tener puntos en el nombre de una unidad (sin contar el del .pas). No tiene nada que nada que ver con la noción de espacio de nombres de .NET ni paquete de Java. Es sólo que '.' tiene el mismo valor que '_' a este nivel, simplemente es parte del nombre.

```
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;
```

Puede observar directamente lo extraordinario del tipo **T**. Pero, ¿es esto realmente un tipo? No, es un parámetro genérico. Con esta clase, si necesito una lista de números enteros, utilizo **TList<Integer>**, ¡en lugar de un "simple" **TList** con muchos operadores necesarios en el código!

Los genéricos nos permiten, por tanto, de alguna manera, declarar una serie de clases (potenciales) de una sola vez, o más exactamente declarar un modelo de clase, pero con uno (o varios) tipos como parámetro, que se pueden cambiar a voluntad. A cada nueva instancia con un nuevo tipo real, es como si estuviera escribiendo una nueva clase real, desde el modelo y, por tanto, otorgarle un verdadero potencial a estas clases.

Como ya he dicho, no voy a detenerme en el por-qué de la forma genérica. Voy a entablar de inmediato su uso sobre una utilidad diaria. Si todavía no entienden, no importa: todo se irá aclarando gradualmente poco a poco.

También puede consultar el tutorial Les génériques sous Delphi .NET **de** Laurent Dardenne.

II – Uso cotidiano de Tlist<T>

Paradójicamente, vamos a ver primero cómo utilizar una clase genérica – es un abuso del lenguaje, deberíamos decir: modelo genérico de clase - a diario, no cómo escribir una nueva. Hay varias buenas razones para ello.

En primer lugar, porque es mucho más fácil diseñar y escribir una clase una vez que tenemos una idea de cómo debemos utilizarla. Y es aún más cierto en cuando descubrimos un nuevo paradigma de programación.

Por otro lado, la mayoría de las presentaciones a la orientación a objetos comienzan en general cómo utilizar una clase creada de antemano.

II –A– Un código simple de base

Para empezar sin problemas, vamos a escribir un pequeño programa que muestra los cuadrados de números enteros de 0 X, dónde X está definida por el usuario.

Tradicionalmente, se utiliza una matriz dinámica (y sería mucho mejor, recuerde que se trata aquí un caso docente), pero vamos a utilizar una lista de números enteros.

Vamos directamente al código:

```
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('Introduzca un número entero :');
    ReadLn(Max);
    WriteSquares(Max);
  except
    on E:Exception do
      Writeln(E.Classname, ': ', E.Message);
  end;

  ReadLn;
end.
```

¿Qué diferencia aprecia usted en este código?

En primer lugar, por supuesto, la declaración de la variable **List** y la creación de la instancia. Al nombre del tipo **TList**, hemos adjuntado en un parámetro real (o su equivalente: es un tipo, no un valor) entre los signos < y >. (En Inglés, se les llama *angle brackets*, lo que significa literalmente: paréntesis en forma de ángulos.)

Usted podrá apreciar pues, que para utilizar una clase genérica, es necesario, asignarle un nombre e indicar un tipo de parámetro. Por ahora, vamos a utilizar aquí siempre un tipo real.

Algunos idiomas permiten un genérico, lo que se conoce el tipo de inferencia, que es para que el compilador adivine el tipo (que se llama parámetro de tipo). Este no es el caso Delphi para Win32. Esa es la razón por la que no se puede escribir:

```
var
  List: TList<Integer>;
begin
```

```
  List := TList.Create; // falta <Integer> aquí
  ...
end;
```

Éste segundo asunto es más importante y menos visible, ya que se trata de una *ausencia*. De hecho, no es necesaria la conversión de enteros en punteros, ¡y viceversa! ¿Práctico, no? Y, sobre todo, mucho más legible.

Además, la seguridad del tipo está mejor gestionada. Con las conversiones, siempre se corre el riesgo de equivocarse, y añadir un puntero a una lista destinada a contener enteros, o viceversa. Si utiliza correctamente los genéricos, usted probablemente casi nunca necesitará las conversiones y, por tanto, cometerá menos errores. Además, si intenta añadir un tipo **Pointer** a una clase de objeto **TList<Integer>**, es el compilador quien le mande de paseo. Antes de los genéricos, ¡un error de este estilo podría no haber sido revelado más que por un valor aberrante, ¡después de su explotación!

Tenga en cuenta que he elegido el tipo **Integer** para limitar el código que no esté directamente vinculado a la noción de genéricos, pero podríamos haber utilizado *cualquier tipo* en su lugar.

II –B– Asignaciones entre clases genéricas

Como usted sabe, cuando hablamos de herencia, podemos asignar una instancia de una subclase a una variable de una clase madre, pero no a la inversa. ¿Qué pasa en los genéricos con esto?

¡Parta del principio de que usted no puede hacer nada! Todos estos ejemplos son falsos, y no compilan:

```
var
  NormalList: TList;
  IntList: TList<Integer>;
  ObjectList: TList<TObject>;
  ComponentList: TList<TComponent>;
begin
  ObjectList := IntList;
  ObjectList := NormalList;
  NormalList := ObjectList;
  ComponentList := ObjectList;
  ObjectList := ComponentList; // sí, hasta esto es falso
end;
```

Como el comentario señala, esto no es porque un **TComponent** puede ser asignados a una **TObject** que un **TList<TComponent>** pueda ser asignados a un **TList<TObject>**. Para entender por qué, piense que **TList<TObject>.Add (Value: TObject)**, permitiría, si la asignación fuera validada, insertar un tipo de valor **TObject** en una lista de **TComponent**!

El otro punto importante a precisar es que **TList<T>** no es en modo alguno especialización de **Tlist,** o una generalización de la misma. De hecho, son dos tipos totalmente diferentes, el primero declarado en la unidad **Generics.Collections**, ¡y el segundo en la unidad **Classes**!

Veremos más adelante en este tutorial, es posible hacer ciertas cesiones a través de las restricciones de los parámetros genéricos.

II-C- Los Métodos de Tlist<T>

Es interesante constatar que **TList<T>** no ofrece el mismo conjunto de métodos que **TList**. Encontramos más métodos de tratamiento de más " alto nivel" y menos de los métodos de "bajo nivel" (como **Last**, que ha desaparecido).

Los nuevos métodos son:

- 3 versiones sobrescritas de **AddRange**, **InsertRange** et **DeleteRange** : son las equivalentes de **Add/Insert/Delete** respectivamente para una serie de elementos ;
- Un método **Contains** ;
- Un método **LastIndexOf** ;
- Un método **Reverse** ;
- 2 versiones sobrescritas de **Sort** (el nombre no es nuevo, pero el uso es diferente) ;
- 2 versiones sobrescritas de **BinarySearch**.

Si llamo su atención sobre estos cambios que pueden parecer insignificantes, es porque son muy característicos del cambio en la forma de trabajo que aportan los genéricos.

¿ Qué interés, de hecho, tendríamos en implementar un método **AddRange** como en la ahora obsoleta **Tlist** ? Ninguno, ya que habría sido necesario la conversión de cada uno de los elementos, a su vez, por lo que sería necesario de todos modos escribir un bucle para construir la matriz a insertar. Y volver a llamar a **Add** directamente en cada vuelta del bucle, a fin de llamar a todos los elementos.

Mientras que con el genérico, tan sólo es necesario escribir el código una vez, y es realmente válido, en su totalidad, para todos los tipos.

Es importante tener en cuenta y comprender esto, es el uso de los genéricos lo que nos permite factorizar mucho mas comportamientos en la escritura de una sola clase.

II –D– Tlist<T> y los comparadores

Es cierto que **TList<T>** puede trabajar con cualquier tipo. Pero, ¿cómo pueden saber cómo comparar dos elementos? ¿Cómo puedo saber si son iguales, siguiendo una búsqueda con un **indexOf**? ¿Cómo saber si uno es menor que el otro, para ordenar la lista?

La respuesta proviene de la comparación. Una comparación es un tipo de objeto **IComparer<T>**. Pues sí, sigue siendo en su totalidad genérico. Este tipo se define en el **Generics.Defaults.pas**.

Al crear una **TList<T>**, puede pasar como parámetro al constructor un comparador, que será utilizado por todos los métodos que necesitemos. Si no lo hacemos, será utilizado el comparador por defecto.

El comparador utilizado por defecto depende del tipo de elementos, por supuesto. Para conseguirlo, **TList<T>** llama al método de clase predeterminado **TComparer<T>.Default**. Este método realiza el trabajo, un tanto bárbaro, basándose en la RTTI para obtener la mejor solución posible de la que es capaz. Pero no es siempre el más adecuado.

Puede mantener el comparador por defecto para los tipos de datos siguientes:

- Los tipos ordinales (enteros, caracteres, lógicos (booleanos), enumeraciones);
- Los tipos comaflotante;
- Los tipos conjuntos (únicamente por la igualdad);
- Los tipos de cadena larga Unicode (string, unicodeString y WideString);

- Los tipos de cadena larga ANSI (AnsiString), pero sin código de página para los '<' y '>';
- Los tipos variante (únicamente para la igualdad);
- Tipos de clase (para la igualdad - usos TObject.Equals);
- Los tipos puntero, y la meta-clase e interfaz (únicamente para la igualdad);
- Tipos de matriz estática u ordinales dinámicos, reales o conjuntos (únicamente para la igualdad).

Para todos los demás tipos, el comparador por defecto realiza la comparación estúpida y desagradable del contenido en memoria de la variable. Es mejor entonces escribir un comparador persolanlizado.

Para ello, hay dos métodos sencillos. Uno de ellos es escribir una nueva función, y el otro sobre la derivación de la clase **TComparer<T>**. Vamos a ambas opciones para la comparación de objetos **TPoint**. Consideraremos que lo que la organizará los puntos es su distancia al centro - hasta el punto (0, 0) - con el fin de tener un orden total (en el sentido Matemático del termino).

II –D- Escribiendo un derivado de TComparer<T>

Nada más sencillo, ¡habéis hecho siempre esto! Un solo método a sobrescribir: **Compare**. Este debe devolver 0 en caso de igualdad, un número estrictamente positivo si el parámetro izquierdo es superior al derecho, y un número estrictamente negativo en el caso contrario.

Veamos lo que esto da:

```
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;
```

Destaque el paso a heredar de TpointComparer, este hereda **TComparer<TPoint>.** Vea pues que es posible heredar una clase "simple" de una clase genérica, a condición de que se le proporcione un valor real para el parámetro genérico.

Para utilizar nuestro comparador, basta con crear una instancia y pasarla a la lista del constructor. Aquí está este pequeño programa que crea 10 puntos al azar, los ordena y muestra la lista ordenada.

```
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; // utiliza el comparador asignado al constructor

    for Item in List do
      WriteLn(Format('%d'#9'%d'#9'(distancia al centro = %.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.
```

¿Y dónde está la liberación de la instancia al comparador en todo esto? Simplemente en el hecho que **TList<T>** toma como comparador una interfaz del tipo **TComparer<T>.** Así es cómo la ordenación de referencia se aplica (implementada en **TComparer<T>**). Por lo tanto, no hay necesidad de preocuparse por la vida, ni la liberación del comparador. Si usted no tiene 'ni idea' del funcionamiento de las interfaces en Delphi, le será muy beneficioso poder consultar el tutorial Les interfaces d'objet sous Delphi **de** Laurent Dardenne.

II-D-2. Escribir un comparador con una función simple de comparación

Esta alternativa parece más simple, según su nombre: no es necesario jugar con más clases. Sin embargo, las presento en segundo termino, porque introducen un nuevo tipo de datos disponibles en Delphi 2009. Se trata de las *referencias* de rutina.

¿En serio? ¿Las conoce? No, no las conoce ;-) Lo que ya conocemos son los tipos de procedimiento, declarados como por ejemplo **TNotifyEvent**:

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

Los tipos de referencia habituales se declaran, en este ejemplo, como TComparison<T>:

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

Hay por lo menos tres diferencias entre los tipos de procedimiento y los tipo referencia de rutina.

La primera es que una referencia de rutina no puede marcarse como **of objeto**. En otras palabras, nunca podremos asignar nunca un método a una referencia a una rutina, sólo ... Rutinas. (O al menos, todavía no he logrado hacerlo ^ ^).

La segunda es más fundamental: mientras que un tipo procedimiento (no **of object**) es un puntero a la dirección de una rutina base (su punto de entrada), un tipo de referencia de rutina ¡es en realidad una interfaz! Con el recuento de referencias y cosas así. Sin embargo, usted no deberá preocuparse nunca, porque su uso diario es idéntico a la de un tipo procedimiento.

La última es la que explica la aparición de referencias de rutina. Se puede asignar una rutina anónima - vamos a ver inmediatamente a que se parece – a una referencia de rutina, pero no a un tipo procedimiento. Pruebe, verá que no compila. Por cierto, esto también explica por qué las referencias de rutina están implementadas como interfaces, pero adentrar en esto está fuera del alcance de este tutorial.

Volvamos a nuestros tres puntos. Para crear un comparador sobre la base de una función, se utiliza otro método de la clase de **TComparer<T>**; se trata de **Construct**. Este método de base toma como parámetro una de referencia de rutina del tipo **TComparison<T>**. Como ya se ha señalado, el uso de las referencias de rutina es muy similar a la de los tipos de procedimiento: se puede utilizar el nombre de la rutina parámetro, directamente. Esto es lo que sucede:

```
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; //utiliza el comparador pasado al constructor

    for Item in List do
      WriteLn(Format('%d'#9'%d'#9'(Distancia al centro = %.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.
```

La única diferencia, por supuesto, es, por lo tanto, la creación del comparador. Mientras que el resto de la utilización de la lista es idéntico (¡afortunadamente!).

Internamente, la clase **Construct** crea una instancia de **TDelegatedComparer<T>**, que toma como parámetro de su constructor la referencia de rutina que se ocupará de la comparación. La llamada a **Construct** devuelve pues un objeto de este tipo, al amparo de la interfaz **IComparer<T>**.

Bueno, finalmente resulta muy simple. De hecho, debemos reconocer esto: ¡ los genéricos están aquí para facilitarnos la vida !

Pero como añadí antes, se puede asignar una rutina anónima a una referencia de rutina. Vamos a ver como resulta:

```
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));

  // Todo igual a continuación...
end;
```

Esta forma de creación es especialmente interesante si este es el único lugar donde usted necesita la rutina de comparación.

Hablando de rutinas anónimas: sí, estas pueden acceder a las variables locales de la rutina / método inclusivo. Y sí, ella puede hacerlo aún después del regreso de esta rutina / método inclusivo. El siguiente ejemplo muestra cómo:

```
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)));

  //siempre lo mismo a continiación
```

```
end;
```

Interesante, ¿verdad?

Llegamos al final de esta pequeña vuelta por los comparados utilizados con **TList<T>**, y con ella la introducción a los genéricos a través del uso de esta clase. En el próximo capítulo, vamos a empezar a ver cómo podemos escribir nuestra propia clase genérica.

III. Concepción de una clase genérica

Ahora que sabemos cómo usar las clases genéricas en el día a día, es el momento de ver cómo funciona una clase genérica en su interior.

Para ello desarrollaremos una clase **TTreeNode<T>**, que será una implementación genérica de un árbol. Esta vez, no es en absoluto un ejemplo docente. Es una clase verdadera que podrá utilizar en sus proyectos reales.

III-A. Declaración de la clase

Comencemos por el principio: la declaración de la clase. Como se puede ver en el extracto de la declaración **TList<T>**, se indica un parámetro genérico (tradicionalmente **T** cuando no tiene ningún significado exacto) entre las signos <>. Por lo tanto, tendremos:

```
unit Generics.Trees;

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

Una rama del árbol se etiquetará con un valor de tipo **T**, y tendrá de 0 a más hijos. Cuando se destruye una rama, o nodo, este liberan a todos sus hijos.

Cómo declaramos un valor del tipo **T**, pues es simple, se declara **FValue: T**; de la misma forma que cualquier otro tipo normal. Para mantener la lista de los hijos, utilizaremos un **TObjectList<U>**, declarado en **Generics.Collections**. He utilizado deliberadamente **U** aquí, porque no debemos confundirnos. ¡ De hecho, **U** será un **TTreeNode<T>** ! Pues sí, se puede utilizar como parámetro de tipo genérico real a una clase genérica.

No implementamos aquí un árbol de búsqueda o de clasificación. Por lo tanto, no necesitamos un comparador como con **TList<T>**.

En cuanto a los campos, quedará de este modo:

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

¿Extraño? Si lo piensas bien, no tanto. Se ha dicho antes que un tipo genérico podía ser sustituido por *cualquier tipo*. Entonces, ¿por qué no una clase genérica?

Quisiera llamar su atención sobre el hecho de que, en el nombre **TTreeNode<T>**, **T** es un parámetro genérico no-real (formal, si lo prefiere). Si bien en el interior de la clase, se convierte en una clase verdadera! Es exactamente como los parámetros de una rutina. En la descripción de la rutina, son parámetros formales, pero una vez en el cuerpo de la rutina, son variables locales como las demás. Esta es la razón por la que se puede utilizar **T** como tipo de **FValue**, o incluso como tipo real para parametrizar **TTreeNode<T>** en la declaración de **FChildren**.

Cuando he dicho que podíamos usar cualquier tipo como parámetro real, les he mentido. Esto no es siempre cierto. Por ejemplo, en **TObjectList<T>**, **T** debe ser un tipo de clase. Es porque **TObjectList<T>** plantea una restricción en su tipo de parámetro. Vemos esto con más detalle en otro capítulo, pero debo señalarlo aquí porque usamos esta clase. Lo utilizamos porque queremos optar a beneficiarnos de la liberación automática de los objetos contenidos por **TObjectList<T>**, lo que no hace **TList<T>**.

Como métodos, además del constructor y destructor, les propondremos métodos de recorrido en profundidad (prefijo y sufijo), así como los métodos para añadir / mover / borrar el nodo. De hecho, la adición se solicitará por el mismo hijo, cuando se le asigne un padre.

Para estos recorridos, necesitamos un tipo de call-back. Usaremos un tipo de referencia de rutina. Y vamos a crear dos versiones de cada recorrido: un recorrido por los nodos, y un recorrido sobre los valores de los nodos. El segundo será el utilizado con más frecuencia cuando se explote la clase **TTreeNode<T>**. La primera es para permitirnos ser más generales, y será utilizada por varios métodos de **TTreeNode<T>** (incluida la segunda versión del recorrido).

Por último, por supuesto tendremos las propiedades para acceder a los padres, los hijos y el valor etiquetado. Lo que da el siguiente código. Usted puede ver no hay nada demasiado nuevo, aparte del hecho de que hay un montón de **T** en todas partes.

Aquí está el texto completo de la declaración de la clase - asegúrese, se lo doy después de haberlo implementado completo ^ ^.

```
type
  /// Referencia a una rutina de call-back con un parámetro
  TValueCallBack<T> = reference to procedure(const Value: T);

  {*
    Estructura del árbol genérica
  *}
  TTreeNode<T> = class(TObject)
  private
    FParent: TTreeNode<T>;                 /// Nodo Principal (nil si es raiz)
    FChildren: TObjectList<TTreeNode<T>>; /// Lista de hijos
    FValue: T;                             /// valor etiquetado

    FDestroying: Boolean; /// Indica si está en trámites de destrucción

    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;
```

Identifiquemos lo que aún es importante aquí. De hecho, no mucho. Si no es que a cada vez que un parámetro es del tipo **T**, se declara como **const**. En efecto, no sabemos de antemano lo que podrá contener **T**. Y hay opciones de sea un tipo "pesado" (como un **string** o un **record**), por lo que es más eficiente utilizar **const**. Ya que **const** nunca esta penalizado (no tiene ningún efecto sobre los tipos "ligeros" como Integer), se utiliza siempre que trabajemos con tipos genéricos.

El concepto de tipo pesado o ligero es un concepto que me es propio, y nada oficial, por eso utilizo las comillas. Entiendo por tipo pesado un tipo cuyos parámetros ganan (en la velocidad de ejecución) al ser pasados como **const** siempre que sea posible. Se trata, en general, de tipos que abarcan más de 4 octetos (excluyendo flotantes y Int64), y los que requieren *inicialización*. Como (muy) gruesos, retengamos las cadenas, los **record**, las matrices, interfaces (no clases) y los Variant.

Sin embargo, cuando un parámetro es del tipo **TTreeNode<T>**, no añadimos **const**. De hecho, lo que será **T**, **TTreeNode<T>** quedará como un tipo de clase, que es un tipo ligero.

III-B. Implementación de un método

Para mostrar las particularidades de la implementación de un método de clase genérico, trabajaremos sobre GetRootNode, que es muy simple. Se escribe de la siguiente forma:

```
{*
  Nudo raíz del árbol
  @return Nudo raíz del árbol
*}
function TTreeNode<T>.GetRootNode: TTreeNode<T>;
begin
  if IsRoot then
    Result := Self
  else
    Result := Parent.RootNode;
end;
```

La único importante a considerar aquí es que, a cada aplicación de un método de una clase genérica, debe especificarse de nuevo la **<T>** junto al del nombre de la clase. En efecto, esto es necesario ya que **TTreeNode** podría perfectamente existir en otros lugares, y, por tanto, confundirse con otro identificador.

III-C. La pseudo-rutina Default

Para implementar los dos constructores que no tienen parámetro **Avalue**, sería preciso iniciar **FValue** con un valor. Sí, pero como no conocemos el tipo de este valor, ¿ cómo escribiremos un valor por defecto, válido para todos los tipos posibles ?

La solución pasa por la nueva pseudo-rutina **Default**. Al igual que **TypeInfo**, esta toma como parámetro un identificador de tipo. Y el nombre de un tipo genérico es también un identificador de tipo.

Este pseudo-rutina "devuelve" el tipo predeterminado por defecto. Podemos escribir:

```
{*
  Crea un nodo con un padre, pero sin valor etiquetado
 @param Aparent el Padre
*}

constructor TTreeNode<T>.Create(AParent: TTreeNode<T>);
begin
  Create(AParent, Default(T));
end;

{*
  Crea un nodo sin padre ni valor etiquetado
*}
constructor TTreeNode<T>.Create;
begin
  Create(nil, Default(T));
end;
```

En el caso de la inicialización de un campo objeto, no es estrictamente necesario. Ya que la inicialización de un objeto arranca de por si todos sus campos a los valores predeterminados por defecto. Pero no he encontrado un lugar mejor para que presentarle esta pseudo-rutina.

III-D. Las referencias de rutina en el recorrido del árbol

Para ilustrar el uso de rutina de las referencias de rutina al otro lado del espejo, e aquí un comentario sobre la implementación del recorrido del árbol.

Cada una de las dos sobrecargas del recorrido toma como parámetro una referencia de rutina de rellamada (comúnmente llamado rutina de call-back). Tenga en cuenta que, también en este caso, hemos utilizado un parámetro **const**. De hecho, el lector atento recordará que las referencias de rutina son interfaces, en realidad. Sin embargo, una interfaz es un tipo pesado. Es necesario utilizar **const** siempre que sea posible.

Aparte de esta pequeña consideración, no hay nada especial que decir acerca del recorrido por los nodos, vea pues:

```
{*
  Recorrido prefijado sobre los nodos del árbol
  @param Action   Acción a realizar sobre cada nodo
*}
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;
```

Para llamar a este call-back, escribimos exactamente lo mismo que con un tipo procedimiento, la misma forma que una llamada de rutina, pero con la variable del tipo referencia de rutina en lugar del nombre de la rutina .

Para implementar la segunda versión, utilizamos la primera, transmitiendo en el parámetro **Action**... ¡Una rutina anónima!

```
{*
  Recorrido prefijado sobre los nodos del árbol
  @param Action   Acción a realizar sobre cada valor
*}
procedure TTreeNode<T>.PreOrderWalk(const Action: TValueCallBack<T>);
begin
  PreOrderWalk(
    procedure(const Node: TTreeNode<T>)
    begin
      Action(Node.Value);
    end);
end;
```

Cuanto menos, es elegante como estilo de Programación, ¿no?

Van a decir ustedes que me repito con mis **const**, pero bueno: el parámetro **Node** de la rutina anónima fue declarado como **const** cuando es claramente del tipo clase. Es que debe ser compatible todavía con la matriz del tipo **TValueCallBack<TTreeNode<T>>,** que solicita que su parámetro sea **const** ;-).

III-D-1. ¿ Hay más métodos que utilizan las rutinas anónimas ?

Pues sí, dos más. **DoAncestorChanged**, cuya misión es hacer un recorrido prefijado sobre el método **AncestorChanged**. Y **BeforeDestruction**, que debe hacer un recorrido prefijado sobre el método de **Destroying** de todos los hijos. Es siempre del mismo modo:

```
{*
  Llama al procedimiento AncestorChanged en todos sus descendientes (Prejifado)
*}
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. Y el resto

El resto, pues bueno ... Es clásico ;-) Nada nuevo. No me voy a eternizar sobre él, pero la fuente completa está disponible para su descarga, al igual que todas las demás, al final del tutorial.

III-F. ¿ Cómo utilizar TTreeNode<T> ?

Es hermoso escribir un árbol de la clase genérica, pero ¿cómo usarlo?

Después de una descripción a lo largo y ancho de la clase **TList<T>**, no hay mucho que decir. Así que me limitaré a darle el código de un pequeño programa de prueba:

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

    // Mostrar árbol con un recorrido prefijado

    Tree.PreOrderWalk(
      procedure(const Node: TTreeNode<Integer>)
      begin
        Write(StringOfChar(' ', 2*Node.Depth));
        Write('- ');
        WriteLn(Node.Value);
      end);
  finally
    Tree.Free;
  end;
end;
```

IV. Concepción de un record genérico

Es hora de ir por otro lado, desarrollemos el **record** simple pero útil **TNullable<T>**. Su propósito es contener un valor sea del tipo **T**, o **nil**. Es muy probable que usted, alguna vez, haya necesitado un tipo así, por ejemplo, para representar el valor NULL de las bases de datos.

Este **record** contendrá dos campos: **FValue** tipo **T** y **FIsNil** de tipo **Boolean**, así como dos propiedades que nos permitan la lectura (pero no la escritura) de estos campos. Utilizaremos únicamente operadores de conversión implícitos para construir valores de este tipo.

```
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;
```


Para obtener más información acerca de la redefinición de los operadores, ver el tutorial Sobrecarga de operadores en de Delphi 2006 para Win32 de Laurent Dardenne.

Es pues un tipo inmutable (No se puede modificador más el estado una vez que se haya creado).

La implementación de tres operadores de conversión es bastante simple. El segundo de ellos (uno con un parámetro de tipo **Pointer**) está aquí para permitir la asignación : = nil.

```
uses
```

```
  SysUtils;

resourcestring
  sCantConvertNil = 'No puede contener nil';
  sOnlyValidValueIsNil = ' El único valor válido es 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;
```

Se puede utilizar muy simplemente como sigue:

```
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;
```

De lo que resulta :

```
5
nil
10
```

¿Lo han entendido todo? Es genial, porque no he dado ninguna explicación. Es la prueba fiel de que son fáciles, los genéricos :-).

El código fuente completo de **Generics.Nullable** está en el zip de fuentes de este tutorial, se puede descargar al final del tutorial.

V. Restricciones sobre los tipos genéricos

Bien, ahora ya saben lo básico. Es hora de ponerse con las cosas serias, a saber: las restricciones.

Como su nombre sugiere, las restricciones permiten imponer limitaciones sobre el tipo real que se

puede utilizar para sustituir a un parámetro de tipo formal. Para continuar la comparación con los parámetros de método: una restricción es al tipo lo que el tipo es a la variable del parámetro. ¿No está claro? Bueno, cuando se especifica un tipo para un parámetro, no podemos transmitirle más que valores que sean compatibles con este tipo. Cuando se especifica una restricción sobre un parámetro de tipo, debe reemplazarlo por un tipo real que responda a estas restricciones.

V-A. ¿ Cuáles son las restricciones posibles ?

No hay más que un número limitado de restricciones posibles. De hecho, sólo hay tres tipos, uno de los cuales todavía no he encontrado su utilidad :-s.

Se puede restringir un tipo genérico a ser:

- Un tipo de clase descendiente de una clase determinada;
- Un tipo interfaz descendiente de una interfaz dada, o un tipo clase que implemente esta interfaz;
- Tipo ordinal, coma-flotante o **record**;
- Un tipo de clase que tenga un constructor sin argumento.

Para imponer una restricción a un parámetro genérico se usa la metodología siguiente:

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

  TIntfListGenericType<T: IInterfaceList> = class
  end;

  TSimpleTypeGenericType<T: record> = class
  end;

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

Lo que impone, respectivamente, que **T** deberá sustituirse por la clase **TStream** o uno de sus descendientes; por **IInterfaceList** o por uno de sus descendientes, por un tipo ordinal, flotante o **record** (un tipo de valor no-nulo, en la terminología de Delphi), o finalmente por una clase que tenga un constructor público sin argumentos.

<T: Class> debe utilizarse en lugar de <T: TObject> ... Se entiende bien que **class** sea aceptado, pero uno se pregunta ¿por qué se rechaza Tobject?

Es posible combinar varias interfaces restringidas, o una clase restringida y una o varias interfaces restringidas. En este caso, el tipo real utilizado debe satisfacer simultáneamente todas las restricciones. La restricción **constructor** también puede combinarse con la restricción de clase y / o de interfaz. Incluso es posible combinar un **record** con uno o varios de interfaz, pero no veo cómo un tipo puede satisfacer ambas al mismo tiempo (en .NET es posible, pero por el momento, no en Win32 !)

Se podría, pero esto no es más que pura especulación por mi parte, que sea en previsión de una versión futura, en la que el tipo Integer, por ejemplo, "implementaría" la interfaz **IComparable<Integer>**. Lo que daría pues un tipo que una cumpliría las restricciones de una declaración como **<T: registro, IComparable<T>>**.

También puede utilizar una clase o interfaz genérico como una restricción, por lo tanto, con un parámetro por especificar. Este puede ser el mismo tipo **T**. Por ejemplo, podemos tratar de imponer que el tipo de elementos debe ser capaz de comparase consigo mismo. Entonces utilizaremos:

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

Si lo que deseamos además es que **T** sea un tipo de clase, podemos combinar:

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

V-B. ¿Pero, para qué sirve?

« Pensaba que el propósito de los genéricos era escribir el código una vez para todos los tipos. ¿Cuál es el beneficio de restringir los posibles tipos? »

Pues bien, esto permite al compilador disponer de más información sobre el tipo utilizado. Por ejemplo, esto le permite saber, con un tipo <T: class>, que es legítimo para llamar al método **Free** residente. O con un tipo <T: IComparable<T>>, que es posible escribir **Left.CompareTo (Right)**.

Para ilustrar esto, vamos a crear una subclase de **TTreeNode<T>**, **TObjectTreeNode<T: class>.** Al igual que **TObjectList<T: class>** propone poner en libertad a sus componentes automáticamente cuando la lista se destruye, nuestra clase deberá liberar el valor de su etiqueta en su destrucción.

De hecho, en realidad es muy poco código, que le expongo de una vez:

```
type
  {*
    Estructura genérica de árbol cuyos valores son objetos
    Cuando el nodo se libera, la etiqueta también se liberada.
  *}
  TObjectTreeNode<T: class> = class(TTreeNode<T>)
  public
    destructor Destroy; override;
  end;

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

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

Eso es todo. El único objetivo es mostrar la técnica. No se trata de ningún truco excepcional.

Hay dos cosas a constatar aquí. En primer lugar, se puede heredar una clase genérica de otra clase genérica, reutilizando de los parámetros genéricos (o no, según convenga).

Luego, en la aplicación de métodos de una clase genérica con restricciones, las restricciones no deben (y no pueden) ser repetidas.

También puede liberar las restricciones y tratar de compilar. El compilador se detendrá en la llamada a **Free**. De hecho, **Free** no está disponible para cualquier tipo. Pero si lo está para cualquier clase.

V-C. Una variante con constructor

También habríamos podido desear que los dos constructores sin parámetros **AValue** de **TObjectTreeNode<T>** creen un objeto para **AValues** en lugar de utilizar **Default(T)** (que, de paso, aquí devuelve **nil** porque T se ve obligada a ser una clase).

Usted puede, como solución, utilizar la llamada **constructor**, como sigue:

```
type
  {*
    Estructura en árbol g genérica cuyos valores son objetos.
    Cuando un nodo se crea sin valor para la etiqueta, un nuevo valor es asignado
    por el constructor sin parámetro del tipo elegido.
    Cuando el nodo se libera, el valor de la etiqueta también es liberado.
  *}
  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;
```

Una vez más, si elimina aquí la restricción **constructor**, el compilador marcará un error sobre el **T.Create**.

VI. Parametrizar una clase con varios tipos

Como habrá podido sospechar, es posible parametrizar una clase con varios tipos. Cada uno, eventualmente, con sus propias restricciones.

Así, la clase **TDictionary<TKey,TValue>** toma como parámetros dos tipos. El primero es el tipo de las llaves, el segundo el tipo de los elementos. Esta clase implementa una tabla hash.

No se equivoque: **TKey** y **TValue** son ambos los parámetros genéricos (formales), no dos tipos reales. No deje engañar por la nomenclatura del código.

La sintaxis de la declaración es un poco laxa sobre este punto. Es posible separar los tipos con comas (,) o por punto y coma (;), eventualmente mezclando los dos cuando hay más de dos tipos. Tanto al nivel de la declaración de la clase, como en el nivel de la implementación de los métodos. En contra, en el uso de un tipo genérico, ¡ debe utilizar siempre comas !

Sin embargo, si coloca una o más instrucciones sobre un tipo que no es el último en la lista, debe utilizar un punto-y-coma para separarla de la próxima. De hecho, una coma indica que habrá una segunda restricción.

Asimismo, permítanme proponerle una norma de estilo - que no es la seguida por Embarcadero. Utilice siempre un punto-y-coma en la declaración del tipo genérico (allí dónde sea muy probable añadir instrucciones) y usar comas en todas las demás ocasiones (implementación de métodos, y utilización del tipo).

Como no tengo mejor ejemplo de tipo genérico para ofrecerle que el mencionado **TDictionary<TKey,TValue>**, le aconsejo que consulte el código de esta clase (definido en la unidad **Generics.Collections**, probablemente tendrá sus propias dudas). Vea tan sólo una muestra :

```
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;
```

Como habra visto, **TDictionary<TKey,TValue>** utiliza nombre de tipos genéricos más explícitos que la **T** que hemos utilizado hasta ahora. Usted debería hacer lo mismo, siempre que el tipo tenga un significado especial, como es aquí el caso. Y, sobre todo, cuando haya más de un tipo en el parámetro.

VII. Otros tipos genéricos

Hasta el momento, no hemos definido más que clases genéricas. Sin embargo, ya nos hemos encontrado con interfaces genéricas (como **IComparer<T>**) y se acaba de cruzar con un tipo **record** genérico (**TPair<TKey,TValue>**).

Por tanto, es plenamente posible definir tanto las interfaces genéricas o tipos **record** genéricos. También es posible declarar una matriz genérica (estática o dinámica), pero dónde sólo el tipo de elementos puede depender de tipos de parámetrizados, pero es poco probable que le encuentre una utilidad real.

No es pues posible declarar un tipo puntero genérico, o un tipo conjunto genérico:

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

VIII. Métodos genéricos

Hasta ahora hemos explorado las distintas posibilidades ofrecidas por los genéricos sobre lo tipos definidos por el desarrollador. Pero también es posible escribir métodos genéricos.

En muchas presentaciones de genéricos o de plantillas para otros lenguajes, esta forma de utilización de los genéricos se presenta en primer lugar. Pero de nuevo, he preferido adelantar lo más útil antes de interesarme en los usos menos frecuentes de los genéricos.

VIII-A. Una función genérica Min

Para introducir el concepto, vamos a escribir un método de clase **TArrayUtils.Min<T>**, que recoge y devuelve el menor elemento de una matriz. Por lo tanto, necesitaremos utilizar un tipo de comparador como **IComparer<T>**.
Al igual que con el nombre del tipo, el nombre del método debe ir seguido de sus parámetros genéricos entre los ángulos. En este caso, el tipo genérico es del tipo de los elementos de la matriz.

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

¡Pues no! No es posible declarar una rutina global con parámetros genéricos. Una posible razón sería el paralelismo con la sintaxis de Delphi.NET para reducir los costes de desarrollo y mantenimiento, internamente.

Para remediar esta falta, se utilizan pues los métodos de clase. Y aún mejor, se precisan como estático. No hay mucho que ver con la palabra-clave del mismo nombre en C++. Se trata aquí de *enlace estático*. Esto quiere decir que este método, no dispone de **Self**, y que su llamada tiene métodos de clases virtuales, o tiene constructores virtuales, no está "virtualizado". En otras palabras, como si no fuera virtual.
Al final, esto hace del método de la clase estático una genuina rutina global, pero con un espacio de nombre diferente.

A diferencia de algunos lenguajes, no es necesario que al menos un parámetro retome cada tipo genérico introducido en el método. Por lo tanto, es posible escribir un método fantasma Dummy<T>(Int: Integer): Integer, que no tiene ningún parámetro formal y cuyo tipo es el *parámetro genérico* **T**. En C++, por ejemplo, esto no sucedería.

Del lado de la implementación del método, es muy similar a las clases. Debemos repetir las escuadras[1] o galones (<>) y los nombres de los tipos genéricos, pero no sus instrucciones eventuales. Esto es pues:

```
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('Ningún registro en la matriz');

  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;
```

Por tanto, nada excepcional ;-)

VIII-B. La sobrecarga y las restricciones

Aprovechemos este ejemplo para revisar nuestras restricciones, y proponer una sobrecarga para los tipos de elementos que soportan la interfaz **IComparable<T>** (esta interfaz se define en **System.pas**).

Y antes de eso, añadiremos otra versión sobrecargada que tiene un tipo de referencia de rutina **TComparison<T>**. Recuerde que usted puede "transformar" un call-back **TComparison<T>** en una interfaz **IComparer<T>** con **TComparer<T>.Construct**.

Usted puede observar la utilización del método **CompareTo** sobre el parámetro **Left**. Esto es posible, por supuesto, porque en esta sobrecarga, el tipo **T** se ve obligado a soportar la interfaz **IComparable<T>**.

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

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

class function TArrayUtils.Min<T>(const Items: array of T): T;
var
```

[1] Nota del traductor: los signos menor ‘<’ y mayor ‘>’, definidos como corchetes angulares (según el termino en inglés) en algún tratado de HTML, no recibe una terminología propia en HTML. El autor utiliza el termino francés ‘Chevrons’ que en mi Larousee se entiende como ‘galón’ (refiriéndose a los signos de rango militar), aunque podríamos interpretar también como ‘corchetes’. Personalmente he preferido el termino **escuadras** ya que dan más la imagen de contenedor, más real con el uso asignado. Me sorprende que en HTML no reciban una nominación más específica.

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

Nótese la llamada **Min<T>:** Es indispensable especificar en la llamada también (o los) el tipo real utilizado. Esto contrasta con otros lenguajes como C++.

Ahora queremos proponer una cuarta versión sobrecargada, siempre con un único parámetro **Items**, pero con un parámetro **T** sin restricciones. Esta versión debe utilizar **TComparer<T>.Default**.

¡ Pero esto no es posible ! Porque, a pesar que las restricciones sobre los tipos cambian, los *parámetros* (argumentos) son los mismos. Así que ambas versiones son completamente ¡ambiguas! Así pues, la siguiente declaración adicional no soporta la compilación :

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

Debemos entonces elegir: abandonar el uno u el otro, o utilizar otro nombre. Y como, hasta que los tipos básicos como Integer soporten la interfaz **IComparable<T>**, corre el riesgo de utilizar tanto el uno primero como el segundo, será preciso pues optar por otro nombre ;-)

```
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;
```

¿ Por qué la conversión explicita en **IComparer<T>** de **Default** que es claramente ya un **IComparer<T>** ? Debido a las referencias de rutina y sobrecargas todavía no están muy bien ensamblados, y el compilador parece lavarse las manos. Sin las conversiones, no soporta la compilación...

VIII-C. Extensiones para TList<T>

Si la clase **TList<T>** es una gran innovación, el hecho es que podría contener más métodos de interés práctico.

Veamos aquí una implementación del método .NET **FindAll** para **TList<T>**. Este método tiene por objeto seleccionar una sub-lista partiendo de una función predicado. La llamada función predicado es una función de rutina call-back que tiene como parámetro un elemento de la lista, y devuelve **True** si lo debemos seleccionar. Se define pues un tipo de referencia de rutina **TPredicate<T>** de la siguiente manera:

```
unit Generics.CollectionsEx;

interface

uses
  Generics.Collections;

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

Entonces, como lamentablemente parece imposible escribir una **class helper** para una clase genérica, vamos pues a escribir un método de clase **FindAll<T>** que realizará esta tarea. Estando privados de **class helper**, vamos a intentar al menos ser más generales y trabajar sobre un numerador cualquiera, con una sobrecarga para un numerable cualquiera.

```
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.
```

Se puede utilizar así:

```
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;
```

Una vez más, la conversión es necesaria debido a la sobrecarga. Es bastante triste, pero es así. Si prefiere no utilizar conversiones, utilice nombres diferentes, o deslíguese de una de las dos versiones.

No depende más que de usted el completar esta clase con otros métodos como este :-)

IX. RTTI y los genéricos

Como último capítulo de este tutorial, vea alguna información sobre lo que ocurre en la RTTI con los genéricos. Si no juega nunca con la RTTI, puede saltarse esta sección por completo. Esto no es en modo alguno una introducción a la RTTI.

IX-A. Los cambios en la Pseudo-ruitna TypeInfo

La RTTI (Run-Time Type Information), siempre se inicia con la pseudo-rutina **TypeInfo**. Usted puede que sepa que no se puede llamar a este pseudo-rutina sobre cualquier tipo; por ejemplo: con los tipos de puntero. Y que, por esto, no devuelve nunca **nil**.

Entonces, ¿ podemos llamar **TypeInfo** sobre un tipo **T** ? La pregunta es pertinente: **T** podría muy bien ser del tipo puntero (invalido para **TypeInfo**), pero igualmente un tipo entero, por ejemplo (valido para **TypeInfo**).

La respuesta es sí, usted puede llamar **TypeInfo** con un tipo genérico. Pero, ¿ qué sucedería entonces si **T** pasa a ser de un tipo que no tiene RTTI ? Bueno, en este caso, y en este caso únicamente, **TypeInfo** devuelve **nil**.

Para ilustrarlo, he aquí un pequeño método de clase que muestra el nombre y la salida de un tipo, pero a través de genéricos:

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

class procedure TRTTI.PrintType<T>;
var
  Info: PTypeInfo;
begin
  Info := TypeInfo(T); // ¡ atención ! Info puede valer nil aquí
```

```
  if Info = nil then
    WriteLn('Este tipo no dispone de RTTI')
  else
    WriteLn(Info.Name, #9, GetEnumName(TypeInfo(TTypeKind), Byte(Info.Kind)));
end;
```

Se utiliza de una forma un tanto particular, en el sentido de que el verdadero parámetro de la rutina **PrintType** se transmite como un tipo parámetrizado.

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

Lo que resulta :

```
Integer  tkInteger
TObject  tkClass
Este tipo no dispone de RTTI
```

IX-A-1. Una función TypeInfo más general

Ya me ha llegado el punto de lamentar que **TypeInfo** no se pueda llamar sobre no importe que tipo, sin recibir **nil**; puede que a usted también. Le presento un pequeño método alternativo que lo hace, sobre la base de los genéricos.

```
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;
```

Lo puede utilizar del siguiente modo :

```
Info := TRTTI.TypeInfo<Pointer>; // Info = nil
// en vez de :
Info := TypeInfo(Pointer); // así obtenemos un error ya que Pointer no tiene RTTI
```

IX-B. ¿ Los tipos genéricos tienen RTTI ?

Hay dos preguntas a plantearse: ¿ Es que los tipos genéricos *nombres instancidos* (con un parámetro **T** no definido) tienen RTTI ? ¿ Y es que los tipos genéricos *instanciados* (donde **T** ha sido sustituido por un tipo real, como **Integer**) tienen RTTI ?

La forma más fácil de averiguarlo es intentarlo ;-) Usted podrá comprobar, testando, que sólo los tipos genéricos instanciados tienen RTTI. De hecho, es bastante lógico, ya que los tipos genéricos no instanciados no son realmente ningún tipo, sólo *modelos* de tipo, y simplemente dejan de existir una vez pasada la compilación.

Podemos preguntarnos qué *nombre* encontraremos para tales tipos. Veamos :

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

Lo que resulta:

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

Esto devuelve por el nombre incluido, entre escuadras*, el nombre completamente clasificado de tipo real reemplazando al tipo genérico.

Puede resaltar que obtenemos **tkInterface** por el tipo de referencia de rutina **TComparison<T>**, lo que demuestra que se trata de una interfaz.

Por lo tanto, estos son los únicos cambios aportados a la RTTI con el advenimiento de los genéricos. No hablo, por supuesto, de otros cambios aportados en esta versión pero que conciernen a las cadenas Unicode.

X. Conclusión

Así concluye el descubrimiento de los tipos genéricos con Delphi 2009. Si nunca ha tenido experiencia con los genéricos en otro lenguaje, esto puede parecer confuso. Pero son realmente muy prácticos, y cambian las vida de los desarrollador, desde que empieza a jugar con ellos.

XI. Descarga de las fuentes

Todas las Fuentes de este tutorial se encuentran en el zip Source sources.zip (Source mirroir HTTP).

XII. Agradecimientos

Un gran agradecimiento a Laurent Dardenne y a SpiceGuid por sus numerosos comentarios y correcciones, que han permitido aumentar la calidad de este tutorial.