Sébastien Doeraene

Generics with Delphi 2009

Себастьян Дура́нэ

Обобщённое программирование (Generics) в Delphi 2009

Автор:

Себастьян Жан Роберт Дуранэ (Sébastien Jean Robert Doeraene) – sjrd.developpez.com

Оригинал учебника:

- *на французском – sjrd.developpez.com/delphi/tutoriel/generiques*
- *на английском – sjrd.developpez.com/delphi/tutoriel/generics*

Перевод:

Алексей Тимохин – TDelphi.blogspot.com

www.tdelphiblog.com/2009/10/generics-delphi-2009-win32.html

Вёрстка, оформление, коррекция, перевод послесловия (X-XII):

Андрей Тишкин – InfoDelphi.ru

Оглавление

Примечания переводчика

Переводить это пособие оказалось не самым приятным занятием. Причина в том, что оригинальный текст был написан на французском, потом кое-как переведён на английский самим автором.

Поэтому большая часть работы над переводом свелась к выяснению того, что же хотел сказать автор набором использованных слов.

Другая часть работы вызвавшая у меня трудности – это адекватный перевод терминов. Буду признателен за замечания и уточнения.

В некоторых местах я просто переписывал текст своими словами, иногда сокращая, иногда дополняя мысль. Так что, по сути, это не перевод, а скорее очень точный пересказ. Ссылки на франкоязычные статьи были заменены ссылками на аналогичные материалы на русском языке.

Терминология:

- **actual types** – фактические типы;
- **anonymous routines** - анонимные методы;
- **callback** – обратный вызов, но я оставил его непереведённым;
- **cast** – приведение типа;
- **class type** – классовый тип, иногда просто класс (так как класс сам по себе является типом);
- **comparer** – компаратор;
- **constraint** – ограничение;
- **generic class** – обобщённый класс;
- **generic parameter** – обобщённый параметр;
- **generics** – дженерики, генерики, параметризованные классы, шаблоны. Мне кажется, что наиболее подходящим для перевода словом в русском языке будет “обобщения”, но в переводе в основном использовал термин дженерики;
- **implicit conversion** - неявное приведение (типа);
- **instanciate** – создание экземпляра объекта;
- **interface type** – интерфейсный тип (данных);
- **ordinal type** – порядковый тип (данных);
- **pre-order walk** – префиксный обход (дерева);
- **prerequisites** – предварительные условия;
- **routine references** - процедурные ссылки;
- **unit** – модуль, юнит;
- типы данных: **string** – строка, **record** – запись.

Алексей Тимохин

I. Введение

I-A. Предпосылки

Это учебное пособие предназначено для программистов с хорошим знанием языка Delphi и требует углублённого понимания принципов объектно-ориентированного программирования.

Для понимания этого пособия предварительные знания о дженериках необязательны. На эту тему уже написана масса статей по другим языкам и я не буду рассматривать в деталях вопросы типа: «Как правильно применять дженерики?». В следующей части читателю будет представлено краткое введение в дженерики, но наиболее полное понимание скорее всего придёт по мере чтения остальных частей пособия и изучения примеров.

Вот некоторые русскоязычные статьи раскрывающие тему дженериков в других языках/платформах, суть везде одна и та же:

- Обобщённое программирование
- Generics в Java 1.5
- Обобщенное программирование под .NET
- Уроки по C++. Урок 29. Использование шаблонов функций

I-B. Что такое дженерики?

Дженерики ещё иногда называют обобщёнными параметрами. Это название чуть лучше раскрывает суть дженериков. В отличие от параметров функций (аргументы), которые имеют значения (values), обобщённые параметры являются типами. И они используются как параметры для классов, интерфейсов, записей или (реже) методов.

Чтобы стало понятнее, вот пример части класса TList<T>, который можно найти в модуле Generics.Collections.pas.

> Да, в имени модулей разрешается использовать точки (кроме .pas, естественно). Это не имеет ничего общего с концепцией имен в. NET, как и с пакетами в Java. Точки - это то же самое, что и _ (символ подчёркивания): просто часть названия.

```
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;
```

Может Вы уже обратили внимание на загадочность типа *T*. Но действительно ли это тип? Нет, это *обобщённый параметр*. Если мне нужен список целых чисел (Integer), то имея такой класс в своем распоряжении, я буду использовать **TList**<Integer>, а не "обычный" **TList**, наряду с кучей приведений типов (casts) в коде!

Дженерики (generics) таким образом позволяют объявить набор (*потенциальных*) классов используя единственный исходный код, или, более официально, *шаблон* класса, который можно использовать с одним или несколькими типами-параметрами, по желанию. Каждый раз, когда инстанциируется новый фактический тип, вы как-будто создаёте отдельный класс, по его шаблону.

Как я сказал ранее, я не буду распространяться на темы того, почему и как работают генерики. Я сразу же перейду к повседневной работе с ними. Если Вам кажется, что у Вас всё ещё нет полного понимания того о чём я говорю, не беспокойтесь: все станет яснее в ближайшее время. Вы также можете прочитать (на французском языке) туториал Les génériques SOUS Delphi. NET, который написал Лоран Дарденн (Laurent Dardenne).

II. Повседневное использование на примере TList<T>

Парадоксально, но мы начнем с примера повседневного использования обобщённых классов (более корректно говорить: обобщённый шаблон класса), вместо дизайна такого типа. Есть несколько веских причин для этого.

Во-первых, значительно легче написать класс, когда у вас есть достаточно четкое представление о том, каким образом вы собираетесь его использовать. Это еще более очевидно в случаях, когда вы открываете для себя новые парадигмы программирования.

Во-вторых, большинство презентаций объектно-ориентированного программирования, как правило, сначала объясняют, как использовать имеющиеся классы.

II-A. Простой код, для начала

Давайте постепенно начнем. Давайте напишем небольшую программу вычисляющую квадраты целых чисел от 0 до X. X будет задаваться пользователем.

Без дженериков, мы бы могли использовать динамический массив (и это было бы гораздо лучше, просто помните о том, что это учебный пример), но мы будем использовать список целочисленных чисел (integer).

Вот код:

```
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.
```

На что здесь стоит обратить внимание?

Прежде всего, конечно, это объявление переменной List, наряду с кодом созданием экземпляра. Мы взяли название обобщённого класса **TList**, и добавили к нему фактический параметр (тип данных, а не значение) между угловыми скобками < и >.

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

Вы можете видеть, что для того, чтобы использовать обобщённый класс, необходимо каждый раз, когда вы пишите его названия, указывать в качестве параметра тип данных. Пока что, мы всегда будем использовать конкретный тип данных.

Некоторые языки позволяют использовать выведение типа (*примечание переводчика:type inference; также смотрите комментарий от Alex Neznanov*), позволяя компилятору угадывать тип параметра. Но в Delphi Win32 такой фокус не пройдёт. Поэтому нельзя писать так:

```
var
  List: TList<Integer>;
  begin
   List := TList.Create; // здесь не хватает <Integer>
   ...
  end;
```

Во-вторых, что более важно, но менее заметно, поскольку речь идёт о том, что фактически отсутствует в коде. А именно cast. Дело в том, что нет необходимости приводить тип Integer к типу Pointer! Удобно, не правда ли? Кроме того, такой код легче читать.

Ещё одно преимущество – это более высокая безопасность при приведении типов при использовании дженериков. Используя приведения типов, всегда остаётся риск совершить ошибку, например, добавив **Pointer** в список, предназначенный для хранения **integer**, или наоборот. Если правильно использовать дженерики, то Вам, скорее всего не придётся прибегать к приведению типов вообще, или всего пару раз. Таким образом, шанс сделать ошибку уменьшается. Более того, даже если попытаться добавить **Pointer** к объекту класса **TList<Integer>**, компилятор откажется компилировать такой код. Без дженериков, такую ошибку можно было бы выявить только по абсурдному значению, может быть даже спустя месяцы после релиза программы.

Я использовал в примере тип Integer, чтобы ограничить код, не связанный непосредственно с представлением дженериков, но на деле, вместо **Integer** можно использовать *любой тип данных*.

II-B. Присвоения между обобщёнными классами

Когда речь заходит о наследовании, как вы знаете, мы можем присвоить экземпляр наследника переменной родительского класса, но не наоборот. Как с этим обстоит дело у дженериков?

Начнём с понимания того принципа, что с дженериками мы не можем сделать ничего подобного! Код в следующем примере являются недействительным и не скомпилируется:

```
var
   NormalList: TList;
   IntList: TList<Integer>;
   ObjectList: TList<TObject>;
   ComponentList: TList<TComponent>;

begin
   ObjectList := IntList;
   ObjectList := NormalList;
   NormalList := ObjectList;
   ComponentList := ObjectList;
   ObjectList := ComponentList; // да, даже это не будет работать
end;
```

Как указано в комментарии, несмотря на тот факт, что **TComponent** являются наследниками от **TObject**, переменная типа **TList<TComponent>** не может быть присвоена переменной типа **TList<TObject>**. Для того чтобы понять почему так происходит, представьте, что если бы присвоение работало, то **TList<TObject>.Add(Value: TObject)** позволило бы добавлять значение**TObject** в список, состоящий из типов**TComponent**!

Другая важная вещь, на которую стоит обратить внимание, что **TList<T>** не является ни особым случаем, ни обобщением от **TList**. На самом деле, это *совершенно разные типы* — первый объявлен в модуле Generics.Collections, а последний в модуле **Classes**!

II-C. Методы TList<T>

Примечательно, что TList<T> не предоставляет тот же набор методов, как **TList**. В нашем распоряжении есть больше методов "высокого уровня", но менее "низкоуровневых " методов (например, таких, как Last, который отсутствует).

Новые методы:

- 3 перегруженные версии методов AddRange, InsertRange и DeleteRange: они эквивалентны методам **Add/Insert/Delete** соответственно, но работают со списками элементов:
  - (const Values: array of T)
  - (const Collection: IEnumerable<T>) · (Collection: TEnumerable<T>)
- Метод Contains;
- Метод LastIndexOf;
- Метод Reverse;
- 2 перегруженные версии **Sort** (это название не является новым, но способ использования отличается);
- 2 перегруженные версии метода BinarySearch.

Я специально обращаю ваше внимание на изменения, которые могут показаться незначительными, потому что именно они очень хорошо иллюстрируют изменения в дизайне, которые привнесли в язык дженерики.

В самом деле, в чём сегодня смысл добавления метода **AddRange** к устаревшему классу **TList** Никакого. Потому что каждый элемент должен подвергаться преобразованию типа. Поэтому, вам всё равно нужно было бы писать цикл, формирующий массив (с преобразованием типов на каждой итерации), для передачи его (массива) в **AddRange**. Вместо этого гораздо проще просто вызывать **Add** в каждой итерации.

Между тем, однажды написанный код с использованием дженериков, остаётся действительно полезным и для других типов данных.

Основная мысль этой главы в том, что дженерики дают огромную свободу при повторном использовании кода.

II-D. TList<T> и компараторы

Конечно, **TList<T>** может работать с любым типом данных. Но вот чего он не знает, это того, каким образом нужно сравнивать два элемента? Как узнать, что они равны, чтобы заработал поиск с помощью **IndexOf**? Как узнать, какой элемент меньше, чем другой, с чтобы упорядочить список?

Ответ – *компараторы*. Компаратор является интерфейсом типа **IComparer<T>.** Этот тип объявлен в **Generics.Defaults.pas.**

При создании экземпляра TList<T>, вы можете передать компаратор в качестве параметра в конструктор, и все методы, которые в нём нуждаются, будут его использовать. В противном случае будет использоваться компаратор по умолчанию.

Используемый по умолчанию компаратор будет зависеть от типа элемента. Чтобы получить его, TList<T> вызывает метод класса **TComparer<T>.Default.** Этот метод с помощью грязных манипуляций с RTTI, пытается получить наиболее подходящее решение, которое, однако, не всегда является подходящим.

Стандартный компаратор можно использовать для следующих типов данных:

- Порядковые *(ordinal)* типы (целые числа, символы, Булевый тип, перечисления);
- Типы с плавающей запятой;
- Множества (для проверки на равенство);
- Длинные юникодные строки (**string**, UnicodeString и WideString);
- Длинные ANSI строки (AnsiString), но *без учёта кодовой страницы*;
- Типы variant (только для проверки на равенство);
- Классы (только для проверки на равенство: используется **TObject.Equals** метод);
- Указатели, мета-классы и интерфейсные типы (только для проверки на равенство);
- Статические и динамические массивы, состоящие из ordinal типов, чисел с плавающей запятой или множеств (только проверка на равенство).

Для всех других типов, компаратор по умолчанию сравнивает только содержимое памяти переменной. Поэтому, для них нужно писать собственный компаратор.

Для этого существуют два простых способа. Первый основан на написание функции, другой на наследовании класса TComparer<T>. Мы проиллюстрируем оба, на примере реализации сравнения для точки (**TPoint).** Предположим, что точки сравниваются согласно их удалённости точки-источника (0, 0), для того иметь возможность провести линейное упорядочивание (в математическом смысле).

II-D-1. Пишем компаратор с использованием TComparer<T>

Это очень просто! Всё что нужно сделать – это переопределить метод: Compare**.** Он должен вернуть:

1. 0 в случае равенства,
2. положительное целое число, если первый параметр больше, чем второй,
3. отрицательное целое если первый параметр меньше второго.

Вот результат:

```
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;
```

Обратите внимание на родителя класса **TPointComparer**: он наследуется от **TComparer<TPoint>**. Как вы видите, вполне возможно наследовать обычный класс от обобщённого класса.

Для того чтобы воспользоваться нашими компаратором, просто создайте экземпляр и передавайте его в конструктор нашего списка.

Вот небольшая программа, которая создает 10 случайных точек, упорядочивает и выводит упорядоченный список:

```
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; // там используется компаратор, который был передан в конструктор

    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.
```

Подождите минуту! А где же освобождается наш компаратор? Напомним, что **TList<T>** принимает *интерфейс* от типа IComparer**<T>**, а не класс. Благодаря механизму подсчёта ссылок для интерфейсов, который корректно реализован в **TComparer<T>**, компаратор будет автоматически освобождён, когда он перестанет быть нужным. Если вы не знаете ничего об интерфейсах в Delphi, я советую прочитать статью Программирование на языке Delphi: Глава 6. Интерфейсы (на русском языке).

II-D-2. Пишем компаратор, используя функцию сравнения

Этот вариант кажется более простым, поскольку как понятно из названия, нет необходимости возиться с дополнительными классами. Тем не менее, я решил представить его вторым, потому он знакомит с новым типом данных, появившимся в Delphi 2009 – процедурными ссылками *(routine references).*

Пытливый читатель сейчас, наверное, захочет возразить, что он уже давно знаком с процедурными ссылками. Хммм… навряд ли. ;-) Скорее всего, читатель подразумевает сейчас процедурные типы *(procedural types)*, например, **TNotifyEvent.**

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

Процедурные ссылки же объявляются по-другому. Вот пример для **TComparison<T>**:

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

Между процедурными ссылками и процедурными типами есть как минимум три отличия:

Во-первых, процедурные ссылки не могут быть методами объекта (помечены **as object).** Иными словами, вы никогда не сможете присвоить процедурной ссылке метод объекта, а только процедуры или функции. (По крайней мере, мне так и не удалось добиться успеха, пытаясь это сделать ^ ^.)

Второе отличие более фундаментально: в то время как процедурные типы (не **методы объекта)** является указателем на базовый адрес процедуры (точку входа процедуры), процедурные ссылки, по сути, являются *интерфейсом!* Со всеми делами, вроде подсчета ссылок и всех остальных штук. Однако, Вас это, вероятно, не будет заботить, поскольку повседневное использование не отличается от использования процедурных типов.

И наконец (и это объяснит появление процедурных ссылок), существует возможность назначить *анонимную процедуру* (мы сейчас посмотрим, на что это похоже) для процедурной ссылки, но не для процедурного типа. Если Вы попробуете так сделать, то получите сообщение об ошибке от компилятора. Кстати, это также объясняет, почему процедурные ссылки реализованы как интерфейсы, но упоминание об этом не входит в рамки этого руководства. *(Примечание переводчика: в оригинале было «Incidentally, that explains also why routine references are implemented as interfaces, but thoughts on this subject are not within the framework of this tutorial.»)*

Давайте же вернемся к нашей сортировке точек. Чтобы создать компаратор на основе функции, мы используем другой метод класса **TComparer<T>,** - *классовую функцию* Construct**.** Этот классовый метод принимает в качестве параметра ссылку на функцию типа **TComparison<T>.** Как было уже сказано, использование процедурных ссылок очень похоже на использование процедурных типов: в качестве параметра можно передавать название процедуры. Вот код:

```
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; // используется компаратор переданный в конструктор

    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.
```

Единственное место, которое изменилось – это создание компаратора. Остальной код использования списка, в целом, остался таким, как и раньше.

Внутри метода Construct создается экземпляр TDelegatedComparer**<T>,** который принимает в качестве параметра своего конструктора ссылку на процедуру (процедурную ссылку), которая будет отвечать за сравнение. Таким образом, вызов Construct вернёт объект того же типа, который инкапсулирован в интерфейсе **IComparer<T>**. *(примечание переводчика: в оригинале было «Calling **Construct** thus returns an object of this type, encapsulated in an interface **IComparer<T>**.»)*

Ну, это тоже очень просто. По сути, мы должны помнить о том, что обобщения (дженерики) предназначены для того, чтобы сделать жизнь легче!

Есть ещё кое-что: существует возможность присвоить анонимную процедуру анонимной ссылке. Давайте посмотрим, как это будет выглядеть:

```
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;
```

Такой способ создания интересен в основном в тех случаях, когда процедура сравнения будет использоваться только в одном месте.

К слову об анонимных процедурах: да, они имеют доступ к локальным переменным той процедуры/метода, в котором они объявлены *(примечание переводчика: в оригинале использовалось слово enclosing - заключены; см. также комментарий)*. И да, они могут делать это даже после того, как процедура или метод, в которой они были объявлены *(примечание переводчика: здесь тоже было enclosing)*, закончил выполнение. Следующий пример проиллюстрирует это:

```
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;
```

Интересно, не так ли?

Вуаля! Вот и подошёл к концу наш небольшой тур по использованию компараторов с TList, а вместе с ним, и введение в генерики на примере использования этого класса. В следующей главе мы начнем изучать, как можно написать свой собственный обобщённый класс.

III. Создание обобщённого класса

Теперь, когда мы знаем, как использовать обобщённые классы, пришло время рассмотреть их внутреннее устройство.

Мы разработаем класс **TTreeNode<T>**, который станет обобщённой реализацией дерева. На этот раз, это будет не просто обучение. Мы создадим реальный класс, который Вы сможете использовать в своих реальных проектах.

III-A. Декларация класса

Давайте же начнем с самого начала: с объявления класса. Как Вы, наверное, помните из объявления **TList<T>,** обобщённый параметр (традиционно называемый **T,** когда нет более точного названия) добавляется между угловыми скобками. Вот так:

```
unit Generics.Trees;
type
  TTreeNode<T> = class(TObject)
  end;
```

Узел дерева будет помечен значением типа **T**, и сможет иметь 0 и более детей. При уничтожении узла, он будет освобождать всех своих детей.

Что за значение типа **T**? Это также просто, как и **«FValue: T;»,** также как и для любого другого нормального типа. Для хранения списка детей, мы будем использовать **TObjectList<U>**, объявленный в **Generics.Collections**. Я использовал здесь **U**, чтобы не запутаться. На деле, **U** будет заменено на **TTreeNode<T>**! Кстати да, разрешается использовать обобщённый класс, как фактический обобщённый параметр.

> Мы не будем реализовывать поиск по дереву. Следовательно, нам не нужно делать компаратор, как мы делали с TList<T>.

Наш класс будет иметь следующие поля:

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

Странно? Если вдуматься, то не очень. Мы уже говорили ранее, что обобщённый тип, может быть заменён *любым другим* типом. Так почему бы не использовать класс обобщённого типа?

> Хочу обратить Ваше внимание на тот факт, что в имени **TTreeNode<T>**,**T** не является фактическим обобщённым параметром (если хотите, формальным параметр). Но внутри класса, он становится фактическим типом! Оно ведёт себя точно также как и параметры процедуры. При объявлении *(примечание переводчика: в оригинале было «In the signature»),* у Вас есть формальные параметры, но в теле процедуры, с ними работают, как и с любыми другими локальными переменными. Вот почему, Вы можете использовать **T** как тип для **FValue**, или даже как актуальный параметр для параметризации **TTreeNode<T>**, при объявлении **FChildren.**
>
> Когда я сказал, чтобы мы можем использовать в качестве фактического параметра любой тип, я был не до конца честен. Это не всегда так. Например, в **TObjectList<T>**, **T** всегда должно быть классовым типом. Это объясняется тем, что **TObjectList<T>** имеет

ограничение на свой параметризуемый тип. Позже мы рассматрим его более подробно, но я хочу указать на него уже сейчас, поскольку именно сейчас мы используем **TObjectList<T>**. А используем мы именно его для того, чтобы извлечь пользу от автоматического освобождения объектов хранящихся в **TObjectList<T>**, чего не умеет делать **TList<T>**.

Помимо конструктора и деструктора, мы предоставим методы для обхода в глубину (префиксного и постфиксного (*примечание переводчика: в оригинале было pre-order и post-order*)), а также методы для добавления/перемещения и удаления дочерних узлов. Собственно, добавление будет запрашивать ребёнок, когда ему будет назначен его родитель.

Для обхода, нам понадобится call-back тип. Угадайте какой? Процедурные ссылки. Мы реализуем два способа *(примечание переводчика: в оригинале, использовался термин "overload" - перегрузка. Но мне это режет слух, поэтому я использовал слово «способ», однако в скобках оставил упоминания об оригинальном термине)* обхода: обход по узлам, и обход по значению узлов. Второй способ (overload), вероятно, будет использоваться чаще при использовании класса **TTreeNode<T>**. Первый же предусмотрен для полноты и является более общим. Он будет использоваться всего в нескольких методах **TTreeNode<T>** (например, во второй перегрузке (overload)).

Наконец, будет, конечно же, доступ к свойствам родителей, детей и значению. Всё это реализуется в коде, приведённом ниже. Как Вы можете заметить, там нет ничего нового, кроме упоминания **T**, где только можно.

Вот полное объявление класса (которое, можете быть уверены, я даю только после того как полностью реализовал класс^^).

```
type
  /// Процедурная ссылка для Call-back с единственным параметром
  TValueCallBack<T> = reference to procedure(const Value: T);

  {*
    Структура обобщённого дерева
  *}
  TTreeNode<T> = class(TObject)
  private
    FParent: TTreeNode<T>;                 /// Родительский узел. (nil для корневого узла)
    FChildren: TObjectList<TTreeNode<T>>; /// Список детей
    FValue: T;                             /// Значение

    FDestroying: Boolean; /// Становится True во время уничтожения объекта

    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;
```

Давайте обозначим, на что здесь стоит обратить внимание. Собственно, не на что, кроме того факта, что каждый параметр типа **T** объявляется как **const.** На момент написания, мы не знаем, чем станет **T**. И соответственно, существует вероятность, что **T** будет одним из «тяжёлым» типов (таким как строка или запись), для которых более эффективно использовать именно **const**. А так как **const** не имеет побочных эффектов (оно просто не даёт эффекта при использовании «лёгких» типов, таких как Integer), мы всегда будем использовать её при работе с обобщёнными типами.

> Концепцию тяжёлых и лёгких типов я придумал сам. Она неофициальная, поэтому никаких цитат и ссылок. Под тяжёлым типом я подразумеваю, тип, чьи параметры (во время исполнения) лучше передавать как **const** всегда, когда это только возможно. По большому счёту, это типы, которые занимают больше чем 4 байта (не считая float и Int64), а также типы, которые требуют *инициализации*. Такие как строки, записи, массивы, интерфейсы (не классы) и Variants.

Однако, если параметр имеет тип **TTreeNode<T>,** мы не будем ставить **Const.** Действительно, независимо от того, чем является **T**, **TTreeNode <T>** остаётся классовым типом, который является «лёгким».

III-B. Реализация метода

Чтобы проиллюстрировать особенности реализации обобщённого метода, давайте рассмотрим простой метод **GetRootNode**. Он написан следующим образом:

```
{*
  Корневой узел дерева
  @возвращает Корневой узел дерева
*}
function TTreeNode<T>.GetRootNode: TTreeNode<T>;
begin
  if IsRoot then
    Result := Self
  else
    Result := Parent.RootNode;
end;
```

Единственное на что здесь стоит обратить внимание, состоит в том, что *каждый* раз, когда Вы пишете реализацию обобщённого метода, необходимо добавлять **<T>** к имени класса. Это обязательное требование, потому что **TTreeNode** (без **<T>**), по сути является другим типом, который кстати может быть объявлен в том же модуле (юните).

III-C. Псевдо-процедура Default

Чтобы реализовать два конструктора, не имеющие параметра **AValue**, нам необходимо как-то инициализировать **FValue**. Естественно, но ведь нам неизвестен фактический тип значения, как мы можем указать значение по умолчанию, которое подошло бы для любого возможного типа?

Выход в добавлении новой псевдо-процедуры **Default**. Как и **TypeInfo**, эта псевдо-процедура будет принимать в качестве параметра идентификатор типа. А название обобщёноого класса, по сути, и является таким идентификатором.

Это псевдо-процедура будет возвращать значение по умолчанию для указанного типа. Поэтому можно будет писать так:

```
{*
  Создаёт узел с родителем и без значения
  @param AParent   Parent
*}
constructor TTreeNode<T>.Create(AParent: TTreeNode<T>);
begin
  Create(AParent, Default(T));
end;

{*
  Создаёт новый узел без родителя и без значения
*}
constructor TTreeNode<T>.Create;
begin
  Create(nil, Default(T));
end;
```

В случае инициализации объектного поля, это, строго говоря, не является необходимым, посколько при создании экземпляра объекта, все его поля *уже* оказываются проинициализированными значениями по умолчанию. Но я не мог найти лучшее место для того, чтобы познакомить Вас с этой псевдо-процедурой.

III-D. Процедурные ссылки и обход дерева

Чтобы взглянуть на процедурные ссылки с другой стороны, давайте посмотрим реализацию обхода дерева.

Оба способа (overloads) обхода принимают в качестве параметра ссылку на процедуру. Обратите внимание, что и здесь мы использовали **const** параметр. Внимательный читатель, наверняка помнит о

том, что ссылки на процедуры, по сути, являются интерфейсами. А интерфейсы мы договорились считать «тяжёлым» типом. Поэтому лучше использовать **const**, когда это возможно.

Кроме этого примера, о первом способе (там, где обход по узлам) сказать особо нечего:

```
{*
  Префиксный обход (preorder walk) узлов дерева
  @param Action   Действие выполняемое для каждого узла
*}
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;
```

Для вызова call-back, мы написали точно такой же код, что и для процедурных типов, т.е. в той же форме, что и вызов процедуры, но используя ссылку на процедуру вместо её имени.

При реализации второго способа (overload), мы воспользуемся первым способом, а в качестве **Action**, будем передавать... анонимную процедуру!

```
{*
  Префиксный обход (preorder walk) по значениям узлов дерева
  @param Action   Действие выполняемое для каждого значения узла
*}
procedure TTreeNode<T>.PreOrderWalk(const Action: TValueCallBack<T>);
begin
  PreOrderWalk(
    procedure(const Node: TTreeNode<T>)
    begin
      Action(Node.Value);
    end);
end;
```

Разве не чудесный кусок кода у нас получился?

> Вы можете сказать, что я непоследователен в том, что говорю о **const** параметрах, так как параметр **Node** в анонимной процедуре объявлен как **const**, хотя это явно классовый тип. Причина в том, что необходимо обеспечить совместимость с сигнатурой типа **TValueCallBack<TTreeNode<T>>**, которая требует именно **const** параметр ;-).

III-D-1. А анонимные процедуры используются в других методах?

Да, ещё в двух. **DoAncestorChanged**, который должен сделать префиксный обход для метода **AncestorChanged**. И **BeforeDestruction**, который должен обойти всех (*do a preorder walk*) детей вызвав метод **Destroying**.

```
{*
  Вызывает метод AncestorChanged для каждого потомка (префиксно)
*}
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. И остальное

В остальном… Это классический пример ;-) Ничего нового. Я не буду распространяться на этот счет, но и полный источник доступен для скачивания, так как все другие, в конце учебника.

III-F. Как использовать TTreeNode<T>?

Это, безусловно, приятно написать класс обобщённого дерева, но как его использовать?

После того как мы рассмотрели в деталях использование обобщённых классов на примере класса **TList<T>**, добавить в сущности нечего. Я только дам Вам код небольшой тестовой программки.

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

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

IV. Создание обобщённой записи

Давайте изучим кое-что ещё и попробуем разработать простенькую, но полезную запись **TNullable<T>**. Идея состоит в том, чтобы создать запись, которая сможет иметь значения типа **T**, либо **nil**. Вы наверное уже задумывались о необходимости такого типа, например для представления значений Null из баз данных.

Эта запись будет содержать два поля: **FValue** типа **T**, и **булевое** поле **FIsNil**. Для чтения полей, созданы два свойства. Мы будем использовать лишь операторы неявного приведения (типов), чтобы построить значения этого типа.

```
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;
```

> За дополнительной информацией о перегрузке операторов я советую обратиться к франкоязычной статье La surcharge d'opérateurs sous Delphi 2006 Win32, которую написал Laurent Dardenne *(примечание переводчика: я не нашёл аналогичного материала на русском языке)*.

Таким образом, это *неизменяемый* тип (объект, чьё состояние не может быть изменено после создания).

Реализация трех операторов преобразования достаточно проста. Второй (с параметром **Pointer**) нужен для того, чтобы можно было делать присвоение : = **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;
```

Использовать эту запись можно так:

```
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;
```

Результат выполнения будет таким:

```
5
nil
10
```

Всё понятно? Это просто чудесно, учитывая, что я не дал ни единого пояснения. Это лишний раз доказывает, что дженерики это также просто как пирог :-). *(примечание переводочника: не знаю почему автор считает что пирог – это просто. Я как-то пробовал испечь один, и скажу честно - программирование намного проще)*

Полный исходный код Generics.Nullable доступен для скачивания в конце учебника.

V. Ограничения обобщенных типов

Хорошо, теперь вы знаете основы. Настало время перейти к серьезным вещам, а именно ограничениям.

Как видно из названия, с их помощью можно ввести некоторые ограничения на фактические типы, которые разрешено использовать для параметризации обобщенного типа. Продолжая аналогию с параметрами функций: ограничение для типа — то же самое, что тип для переменной. Не понятно? Хорошо, вот пример: если Вы укажете тип параметра функции, то Вы можете передать в качестве параметра только выражение, которое будет соответствовать этому типу. А когда Вы указываете ограничения на параметр обобщённого типа, Вы должны будете заменить его фактическим типом, удовлетворяющим этим ограничениям.

V-A. Какие ограничения можно использовать?

Существует не так много ограничений доступных для использования. Точнее, всего четыре вида.

Допустимые обобщённые типы можно ограничить следующими способами:

- классовым типом, который является потомком от заданного класса;
- тип интерфейса, являющимся потомком от заданного интерфейса или классом, реализующим этот интерфейс;
- порядковым (ordinal), числом с плавающей запятой или записью;
- классом, имеющим конструктор без аргументов.

Для того, чтобы наложить ограничение на обобщённый параметр, нужно написать:

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

  TIntfListGenericType<T: IInterfaceList> = class
  end;

  TSimpleTypeGenericType<T: record> = class
  end;

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

Эти синтаксис, соответственно, указывает, что **T** может быть заменен только на:

1. класс **TStream** или один из его потомков;
2. **IInterfaceList** или один из его потомков;
3. на порядковый (ordinal), число с плавающей точкой или запись (один ненулевой тип данных, следуя терминологии Delphi);
4. класс, обеспечивающий конструктор с нулевым числом аргументов.

> Нужно использовать <T: class> вместо <T: tobject>... Понятно, что **класс** будет принят, но может возникнуть вопрос, почему отклоняется **TObject**.

Можно комбинировать несколько интерфейсных ограничений или ограничений по классу вместе с одним или несколькими интерфейсными ограничениями. В этом случае, реальный тип должен удовлетворять сразу *всем* ограничениям. Ограничение по **конструктору** могут быть совмещены с

ограничениями по классу и/или ограничениями по интерфейсу. Можно даже объединить ограничение по записи с одним или несколькими ограничениями по интерфейс, но я не мог себе представить реально существующий тип, который бы удовлетворял этим двум ограничениям сразу (в NET такое возможно, но не в текущей реализации Delphi для Win32)!

> Может быть, (но это только моё предположение) это введено в преддверии будущих версий, где Integer, например, будет "реализовывать" интерфейс **IComparable<Integer>**. В таком случае, он вполне сможет удовлетворить ограничению вроде **<T: record, IComparable<T>>**.

Кроме того, можно использовать в качестве ограничения обобщённый класс или интерфейс с заданным типом параметра. Этим параметром может быть само **T**. Например, может потребоваться тип которого можно сравнить с собой (с заданным). Тогда можно было бы написать:

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

Дополнительно, если вы хотите, что бы **T** было классового типа, вы можете написать так:

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

V-B. Зачем нужны ограничения?

"Я думал, что генерики предназначены для того чтобы использовать один код сразу для *всех* типов. Какой смысл в том, чтобы ограничивать используемые типы?"

Ну, ограничения позволяют компилятору получить больше информации об используемых типах. Например, ограничение <T: class>, позволяет компилятору узнать, что для переменной типа **T,** разрешается вызывать метод **Free.** А из ограничения <T: IComparable<T>> сделать вывод, что разрешается использовать **Left.CompareTo(Right)**.

Мы рассмотрим это на примере дочернего класса **TTreeNode<T>**, вызываемого **TObjectTreeNode<T: class>**. Следуя примеру **TObjectList<T: class>**, который обеспечивает автоматическое освобождение его элементов при разрушении, наш класс будет при уничтожении освобождать своё значение (*labelled value*).

На самом деле, там будет не так уж много кода, учитывая, что большую часть мы уже написали в нашем суперклассе.

```
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;
```

Вот и всё. Цель этого примера только в том, чтобы показать, как использовать этот приём, а никак не создать исключительный код.

Обратите внимание на две вещи. Во-первых, обобщённый класс может наследоваться от другого обобщенного класса, используя тот же обобщённый параметр (или без него).

Во-вторых, при реализации методов обобщённых классов с ограничениями, ограничения не должны повторяться (*примечание переводчика: в оригинале было «must not be repeated», что правильнее перевести «запрещено повторять»*).

Можно, кстати, попытаться удалить ограничение и скомпилировать ещё раз. В этом случае компилятор выдаст ошибку на вызове **Free**. Естественно, ведь **Free** доступен не для всех типов, а только для *классов.*

V-C. Вариант с конструктором

Возможно Вы захотите иметь конструктор без параметра **AValue**, чтобы иметь возможность самому создать объект для **FValue** вместо того, чтобы использовать **Default(T)** (последний вернёт **nil** здесь, потому что **T** имеет ограничение только на классы). *(примечание переводчика: я так и не понял, что хотел сказать автор этой фразой. Ссылка на оригинал. Однако, ограничения хорошо описаны в справке к Delphi).*

Ограничение по **Конструктору** предназначена для этой цели, которая дает:

```
type
    {*
      Структура обобщённого дерева, чьими значениями являются объекты
      Когда узел создаётся без значения, новое значение создаётся с помощью
      Конструктора T по умолчанию (конструктора без аргументов).
      Когда узел освобождается, значение также освобождается.
    *}
  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;
```

Опять же, если удалить ограничение по **конструктору**, то компилятор выдаст ошибку на вызове **T.Create**.

VI. Использование в качестве параметра более чем одного типа

Как вы уже, наверное, догадались, при параметризации обобщённого типа, можно использовать несколько типов. Каждый из них может иметь свой собственный набор ограничений.

Таким образом, класс **TDictionary<TKey,TValue>** принимает два типа параметров. Первый тип для ключей, а второй – тип для элементов. Этот класс реализует хеш-таблицу.

> Не стоит заблуждаться: **TKey** и **TValue** в действительности (формально) являются обобщёнными параметрами, а не реальными типами. Не надо смешивать эти вещи из-за нотации.

В этом месте, синтаксис декларации довольно слаб. Здесь возможно отделить типы запятыми (,) или точкой с запятой (;). В случае, когда типов больше двух, даже смешанное использование разделителей будет принято. Как на уровне декларации так и на уровне реализации. Однако, при *использовании* обобщённых типов, вы должны использовать запятые!

При этом, если вы размещаете одно или несколько ограничений по типу, который не является последним, вам придется использовать точку с запятой, чтобы отделить его от следующего. Действительно, использование запятой будет означать дополнительное ограничение.

Итак, позвольте мне предложить Вам стилистическое правило, которое не является тем правилом, которому следуют в Embarcadero. Всегда используйте точку с запятой при объявлении обобщённого типа (в случаях использования ограничений), и используйте запятые в остальных местах (при реализации методов и использовании обобщённых типов).

Поскольку у меня нет лучшего примера обобщённого типа с двумя параметрами, чтобы предложить Вам, кроме TDictionary<TKey,TValue>,я предлагаю Вам взглянуть на код этого класса (объявленного, как Вы, возможно, догадались, в Generics.Collections). Вот выдержка из него:

```
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;
```

> Как Вы уже заметили, **TDictionary<TKey,TValue>** более чётко определённое название для обобщённого типа, чем опостылевший **T**, который мы использовали до сих пор. Я советую Вам делать тоже, если тип имеет определённое значение, как в данном случае. Особенно когда есть несколько параметризуемых типов.

VII. Другие типы дженериков

До сих пор мы рассмотрели лишь обобщённые *классы* и *записи*. Тем не менее, мы уже вплотную приблизились к некоторым обобщённым *интерфейсам*.

Кроме того, существует возможность задекларировать типы обобщённых *массивов* (статические или динамические), где только тип *элементов* может зависеть от обобщённого типа (а не измерения). Но маловероятно, что Вы найдете практическое применение для этого.

Таким образом, не представляется возможным объявить указатель обобщённого типа (generic pointer type), ни множество обобщённого типа (generic set type):

```
type
  TGenericPointer<t> = ^T; // ошибка компилятора
  TGenericSet<t> = set of T; // ошибка компилятора
```

VIII. Обобщённые методы

Мы рассмотрели много возможностей, которые предоставляют дженерики, применительно к типам, определённым разработчиком. Настало время перейти к обобщённым *методам*.

Эту часть часто объясняют во многих презентациях и шаблонах по использованию дженериков в других языках. Но я предпочёл начать с обобщённых типов, которые используются чаще всего.

VIII-A. Обобщённая функция Min

Для того, чтобы лучше представить эту концепцию, мы напишем метод для класса **TArrayUtils.Min<T>**, который найдёт и вернёт самый маленький элемент массива. Для этого нам, конечно же, понадобится компаратор типа **IComparer<T>**.

Как и название обобщённого типа, название обобщённого метода должен сопровождаться обобщённым параметром, расположенным между угловыми скобками. Здесь, обобщённый тип является типом элемента массива.

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

> К сожалению, не представляется возможным объявить глобальную *процедуру* с обобщёнными параметрами. Вероятно, причина этого ограничения связана с параллелизмом в синтаксисе Delphi.NET, чтобы уменьшить усилия по разработке и поддержанию.

> Чтобы компенсировать это ограничение, мы будем использовать классовые методы. Лучше, чтобы они были объявлены с ключевым словом **static**. У него нет ничего общего с омонимичным ключевым словом в C++. Речь здесь идёт о *статическом связывании*. Это говорит о том, что в таком методе нет **Self**, и тогда вызываются виртуальные методы класса, или виртуальные конструкторы должны быть "виртуализированы".(примечание переводчика: я не понял, что здесь имелось в виду: «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".»). Другими словами, это как если бы они больше не были виртуальными. В результате, это приводит к подлинной глобальной процедуре, но в другом пространстве имён.

> В отличие от других языков, параметр не обязательно должен использовать обобщённый тип. Таким образом, можно написать метод Dummy<T>(Int: Integer): Integer, который не имеет *формального* параметра, чей тип зависел бы от *обобщённого параметра* **T**. В C++, например, такое не получилось бы даже скомпилировать.

Реализация этого метода очень похожа на классы. Вы должны повторить угловые скобки для обобщённого типа, но не для возможных ограничений. Наш метод Min будет выглядеть следующим образом:

```
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;
```

Ничего исключительного, как Вы можете видеть. ;-)

VIII-B. Перезагрузка и ограничения

Давайте воспользуемся преимуществом этого прекрасного примера, чтобы пересмотреть наши ограничения и сделать перегруженную версию для тех типов элементов, которые поддерживают интерфейс **<T>** (он объявлен в **System.pas**).

Перед этим, давайте добавим ещё одну перезагрузку, которая будет принимать процедурную ссылку типа **TComparison<T>**. Помните, мы легко можем «преобразовать» функцию обратного вызова типа **TComparison<T>** в интерфейс **IComparer<T>**, вызвав метод **TComparer<T>.Construct**.

Вы можете посмотреть на использование метода **CompareTo** на параметре **Left**. Такой вызов, допускается, только из-за наличия ограничения.

```
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;
```

> Обратите внимание на *вызов* **Min<T>**: указание актуального типа между угловыми скобками является обязательным даже там. Это не относится к другим языкам, таким как C++.

Теперь мы хотим предоставить четвёртый перегруженный метод, снова только с одним параметром **Items**, но на этот раз с параметром **T** без ограничений. Эта версия должна будет использовать **TComparer<T>.Default**.

Но это невозможно! На самом деле, несмотря на то, что ограничения на типы будут измены, параметры (аргументы) остаются теми же самыми. Поэтому, такие перегруженные методы будут совершенно двусмысленны! Таким образом, следующее объявление приведёт к ошибке компилятора:

```
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; // ошибка компилятора
  end;
```

Следовательно, мы должны выбрать: отказаться от первого или второго, или же использовать другое имя. И до тех пор, пока базовые типы, такие как **Integer** не начнут поддерживать интерфейс **IComparable<T>**, Вам могут понадобиться оба, остаётся выбрать использование другого имени;-)

```
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;
```

Почему я явно привожу **Default** к типу **IComparer<T>**, несмотря на то, что он *уже* явно означает **IComparer<T>**? Потому что процедурные ссылки и перегруженные методы до сих пор не очень хорошо работают друг с другом, и компилятор, кажется, начинает путаться от такого смешения. Без приведения типов, компиляция не удаётся...

VIII-C. Некоторые дополнения для TList<T>

Несмотря на то, что **TList<T>** является очень хорошим нововведением, оно никогда не сможет предоставить более практичных методов.

Вот пример реализации .NET метода **FindAll** для **TList <T>.** Этот метод направлен на получение подсписка с помощью функции предиката. Под функцией предиката здесь подразумевается процедура обратного вызова, которая принимает элемент списка, и возвращает **True**, если он должен быть выбран, и **False** в обратном случае. Мы определяем процедурную ссылку типа **TPredicate<T>** следующим образом:

```
unit Generics.CollectionsEx;

interface

uses
  Generics.Collections;
```

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

К сожалению, невозможно написать **class helper** для обобщённого классы, поэтому мы напишем метод класса **FindAll<T>** который реализует это. Поскольку мы лишены возможности использовать **class helper**, мы постараемся хотя бы использовать его преимущество в том, чтобы реализовать более общее решение, работающее с энумератором или перечисляемым типом. (*примечание:* в оригинале было «enumerator or 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.
```

Можно использовать этот метод следующим образом:

```
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;
```

> Опять же, приведение типов необходимо из-за перезагрузок. Это довольно печально, но пока это так. Если вы предпочитаете обходится без приведений типов, используйте разные названия или постарайтесь избавится от одной из двух перезагрузок.

Задачу, дополнить этот класс другими методами наподобие **FindAll** я оставлю вам. :-)

IX. RTTI и дженерики

В этой последней главе, я собираюсь рассказать о том, что генерики изменили в RTTI. Если Вы никогда не играли с RTTI, Вы можете пропустить эту главу. Она не была задумана как введение в RTTI.

IX-A. Изменения в псевдо-процедуре TypeInfo

RTTI всегда начинается с псевдо процедуры **TypeInfo**. Вы наверное знаете, что определённые типы не могут передаваться в эту псевдо процедуру. Например, указатели. И поэтому, она никогда не возвращает **nil**.

Итак, можем ли мы вызвать **TypeInfo** для обобщённого типа **T**? Вопрос уместный: ведь **T** может быть указателем (не валидным для **TypeInfo**) или целым (integer) (валидный).

Ответом будет да, разрешается вызывать **TypeInfo** для обобщённого типа. Но что случится, если **T** окажется типом, который не имеет RTTI? Именно в этом единственном случае, **TypeInfo** вернёт **nil**.

Чтобы проиллюстрировать этот феномен, вот метода небольшого класса, который печатает имя и тип типа, но делает это *с помощью* дженериков:

```
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;
```

Этот метод, довольно специфичен. В том смысле, что его реальный параметр в действительности передаётся как параметризованный тип.

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

Выполнение его даёт результат:

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

IX-A-1. Более общая функция TypeInfo

Я уже пожалел, что **TypeInfo** не может быть применена к любому типу, даже в случае, если результатом будет **nil**; может и вы сделали также. Поэтому, вот небольшая замена этого метода методом, решающим эту проблему.

```
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;
```

Вы можете использовать его так:

```
Info := TRTTI.TypeInfo<Pointer>; // Info = nil
// вместо:
Info := TypeInfo(Pointer); // ошибка компилятора, так как у указателя (Pointer) нет RTTI
```

IX-B. Есть ли RTTI у обобщённых типов?

На самом деле здесь два вопроса: есть ли RTTI у *неопределённых* обобщённых типов (тех, у которых не определён параметр **T**)? И есть ли RTTI у *определённых* обобщённых типов (тех, у которых **T** было заменено реальным типом, таким как **Integer**)?

Самый простой способ узнать это – попробовать самим ;-) Взгляните на простой тест для *определённых* обобщённых типов. На самом деле, это вполне логично, так как *неопределённые* обобщённые типы, на самом деле не являются реальными типами, а скорее *шаблонами* типов, и они перестают существовать, после окончания компиляции.

Можно спросить себя, какое название будет у такого типа. Итак:

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

И результат:

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

Это показывает, что название в себя включает между угловыми скобками полное имя фактического типа, заменяя параметр типа.

> Вы также можете заметить, что тип процедурной ссылки **TComparison<T>** выдаст результат **tkInterface**, и это доказывает то, что это действительно интерфейс.

Вот и всё что я хотел сказать об RTTI и дженериках. Конечно же я не буду здесь рассказывать о других изменениях в RTTI появившихся в Delphi 2009 из-за юникодных строк.

X. Послесловие

Это конец нашего путешествия среди дженериков с Delphi 2009. Если вы ещё не имели возможность поэкспериментировать с дженериками в других языках, оно могло оказать разрушительное воздействие. Но на самом деле они очень практичны и действительно изменяют жизнь разработчика, как только он немного поиграется с ними.

XI. Загрузите исходный код

Исходный код всех примеров этого учебника (документирован на французском), прилагается: Sources.zip

XII. Благодарности

Я хотел бы поблагодарить Лорана Дарденн (Laurent Dardenne) и SpiceGuid за их многочисленные комментарии и исправления, которые позволили значительно улучшить данный учебник.

Выражаю благодарность также Тибо Кувелье (Thibaut Cuvelier) за его корректировки к моему переводу на английский.