

KEEPER'S BLOG

== I'M STARTING WITH THE MAN IN THE MIRROR ==

Используем дженерики в Delphi!

Обобщенное программирование стало доступно Delphi-разработчикам начиная с 2009 версии продукта. По настоящее время они были освещены в различных источниках - в первую очередь, в ежегодной книге М. Кэнту - *Delphi 2009 Handbook* и, конечно, в справочной системе *Delphi*. Предназначение данного материала - максимально доступно познакомить аудиторию с дженериками и показать преимущество их использования на простых и интересных примерах.

Содержание

I. Часть 1 - Введение

1. Что такое дженерики и зачем они нужны?
2. Преимущества использования дженериков
 1. Безопасность типов
 2. Эффективность
 3. Максимальное повторное использование кода
3. Встроенные обобщенные классы в Delphi
4. Что "поддается обобщению" в Delphi?
 1. Обобщенные методы
 2. Обобщенные классы
 3. Обобщенные записи
5. Заключение

II. Часть 2 - Системные классы

1. Введение
2. `Generics.Defaults`
 1. `TComparer<T>`
3. `Generics.Collections`
 1. `TArray`
 1. Сортировка и поиск элементов в одномерном массиве
 2. Сортировка двумерного массива
 2. `TDictionary<T>` и `TObjectDictionary<T>`
 1. Работа со словарем (на примере `TDictionary<T>`)
 2. События `OnKeyNotify` и `OnValueNotify`
 3. Ключи, значения и `TArray`
 3. `TList<T>` и `TObjectList<T>`
 1. Работа со списком (на примере `TObjectList<T>`)
 2. Поиск и сортировка
 3. Событие `OnNotify`
 4. `TStack<T>` и `TObjectStack<T>`
 5. `TQueue<T>` и `TObjectQueue<T>`
4. Заключение

III. Часть 3 - Приложение

1. Заключение
2. Исходники
3. Ссылки

Часть 1 - Введение

1. Что такое дженерики и зачем они нужны?

Наличие обобщений в языке позволяет создавать открытые типы, которые превращаются в закрытые на этапе компиляции. Синтаксис дженериков на примере обобщенной записи `TPoint<T>` приведен в *Листинге 1*:

Листинг 1 - Объявление обобщенной записи TPoint<T>

```
1. type
2.   TPoint<T> = record
3.     X: T;
4.     Y: T;
5.   end;
```

Сразу бросаются в глаза отличия от декларирования обычной записи - наличие `<T>` в имени записи и координат `X` и `Y` этого же типа `T`. `T` здесь - неуточненный тип, который будет указан позже, при создании конкретного экземпляра записи.

Предположим, что мы решили использовать в приложении "дробные" точки (например, `Double`). Все, что нужно сделать - объявить следующий закрытый тип:

Листинг 2 - Использование обобщенной записи TPoint<T> в качестве "дробной" точки

```
01. ...
02.
03. var
04.   MyPoint: TPoint<Double>;
05. begin
06.   MyPoint.X := 1.5;
07.   MyPoint.Y := -0.5;
08.
09. ...
```

А если нам понадобится целый тип, мы просто изменим `Double` на `Integer`:

Листинг 3 - Использование обобщенной записи TPoint<T> в качестве "целой" точки

```
01. ...
02.
03. var
04.   MyPoint: TPoint<Integer>;
05. begin
06.   MyPoint.X := 1;
07.   MyPoint.Y := 100;
08.
09. ...
```

Просто, не правда ли? `MyPoint: TPoint<Double>` и `MyPoint: TPoint<Integer>` - уже являются закрытыми типами и подчиняются все правилам, справедливым для обычных, необобщенных типов.

Может возникнуть вопрос: могу ли я сделать это без дженериков? Конечно, можете. Правда, лишитесь ряда преимуществ.

2. Преимущества использования дженериков

2.1. Безопасность типов

Когда необходимо повысить безопасность типов и избежать ошибок их несоответствия во время выполнения приложения - дженерики могут прийти на помощь. Для демонстрации сравним стандартный класс `TList` и его обобщенный "аналог" `TList<T>`. Как известно, `TList` хранит массив указателей на объекты, причем тип этих объектов может быть различен. Рассмотрим следующий пример:

Листинг 4 - Вызов метода класса TCustomer для элементов TList

```
01. ...
02.
03. type
04. // TCustomer - произвольный класс "Клиент"
05.
06. TCustomer = class
07. private
08.     FName: string;
09. private
10.     constructor Create(const Name: string);
11.     function ShowName: string;
12. end;
13.
14. ...
15.
16. procedure PrintCustomersInfo(List: TList);
17. var
18.     Item: Pointer;
19.     Customer: TCustomer;
20. begin
21.     for Item in List do
22.         ShowMessage((TObject(Item) as TCustomer).ShowName);
23. end;
24.
25. procedure PrintCustomersInfo2(List: TList);
26. var
27.     Item: Pointer;
28.     Customer: TCustomer;
29. begin
30.     for Item in List do
31.         if TObject(Item) is TCustomer then
32.             ShowMessage((TObject(Item) as TCustomer).ShowName);
33. end;
```

Теперь представьте, что передаваемый TList содержит не только экземпляры TCustomer. Для PrintCustomersInfo это будет катастрофично и приведет к Invalid Type Cast, в процедуре PrintCustomersInfo2 мы избежали этого путем дополнительных проверок.

Но разве не замечательно бы было отдать такие проверки на откуп компилятору при сборке приложения? Дженерики позволяют это сделать:

Листинг 5 - Вывод информации о клиентах через TList<T>

```
1. procedure PrintCustomersInfo3(List: TList<TCustomer>);
2. var
3.     Customer: TCustomer;
4. begin
5.     for Customer in List do
6.         ShowMessage(Customer.ShowName);
7. end;
```

Заметили, что код уменьшился и стал более читаемым? Кроме того, за тем, чтобы в TList не попало ничего лишнего уже проследил компилятор.

2.2. Эффективность

Дополнительная эффективность при использовании дженериков - возможно, одно из главных их преимуществ. Обобщения предоставляют компилятору больше информации, не исключая данные о типе во время исполнения приложения. Такой код проще писать, эффективнее заниматься отладкой приложения. Кроме того, в рассматриваемом примере ассемблерный код с дженериками (PrintCustomersInfo3) содержит до 10 инструкций меньше (по сравнению с PrintCustomersInfo2).

2.3. Максимальное повторное использование кода

Обобщенный класс, код для которого был написан всего 1 раз, может использоваться многократно. Так, без переписывания кода, `TList<T>` может быть использован для создания списка дробных чисел (`TList<Integer>`), строка (`TList<string>`) и т.д.

В любом случае, эти преимущества достаточно существенны для того, чтобы пользоваться ими в полной мере.

3. Встроенные обобщенные классы в Delphi

"Из коробки" в Delphi уже имеется ряд стандартных обобщенных классов, которые можно использовать при написании приложений. Находятся они в модулях `Generics.Defaults` и `Generics.Collections`.

Основные классы и типы данных приведены в *Таблицах 1 и 2*.

Таблица 1 - Некоторые классы модуля `Generics.Defaults`

<code>IComparer</code>	Обобщенный интерфейс <code>IComparer</code> предназначен для сравнения двух значений одинакового типа
<code>IEqualityComparer</code>	Обобщенный интерфейс <code>IEqualityComparer</code> используется для проверки равенства двух значений
<code>TComparer</code>	Базовый обобщенный класс для классов, реализующих интерфейс <code>IComparer</code>
<code>TEqualityComparer</code>	Базовый обобщенный класс для классов, реализующих интерфейс <code>IEqualityComparer</code>
<code>TCustomComparer</code>	Базовый обобщенный класс для классов, реализующих интерфейсы <code>IComparer</code> и <code>IEqualityComparer</code>

Таблица 2 - Некоторые классы и типы модуля `Generics.Collections`

Классы	
<code>TArray</code>	Класс, содержащий статические методы для поиска и сортировки обобщенного массива
<code>TDictionary</code> , <code>TObjectDictionary</code>	Словарь (коллекция пар ключ-значение)
<code>TList</code> , <code>TObjectList</code>	Упорядоченный список
<code>TStack</code> , <code>TObjectStack</code>	Реализация стека (последний пришел, первый вышел)
<code>TQueue</code> , <code>TObjectQueue</code>	Реализация очереди (первый пришел, первый вышел)
Типы	
<code>TPair</code>	Запись, хранящая пару ключ-значение
<p>Примечание: как и аналоги из модуля <code>Classes</code>, обобщенные "объектные" классы относительно "необъектных" (например, <code>TObjectList<T></code> по сравнению с <code>TList<T></code>) позволяют хранить объекты в качестве своих элементов, а также автоматически следить за их жизненным циклом</p>	

Использовать стандартные обобщенные классы довольно просто: включаем соответствующие модули в раздел `uses` и задействуем нужные нам классы. В *Листинге 6* приведен пример работы со списком целых чисел на основе обобщенного класса `TList<T>`.

```
01. ...
02.
03. uses
04.     Generics.Collections;
05.
06. ...
07.
08. var
09.     IntegerList: TList<Integer>;
10. begin
11.     IntegerList := TList<Integer>.Create;
12.     try
13.         IntegerList.Add(1);
14.         IntegerList.Add(5);
15.         IntegerList.AddRange([2, 5, 8, 9]);
16.         IntegerList.Insert(0, 0);
17.         // Имеется ли 9 в списке = True
18.         ShowMessage(BoolToStr(IntegerList.Contains(9), True));
19.         IntegerList.Remove(9);
20.         // Теперь 9-ки уже нет = False
21.         ShowMessage(BoolToStr(IntegerList.Contains(9), True));
22.         // Индекс двойки в списке равен 2? True
23.         ShowMessage(BoolToStr(IntegerList.IndexOf(5) = 2, True));
24.     finally
25.         FreeAndNil(IntegerList);
26.     end;
27. end;
```

Более подробно системные классы будут рассмотрены во 2-м разделе.

4. Что "поддается обобщению" в Delphi?

Естественно, что в Delphi имеется возможность не только использовать имеющуюся библиотеку дженериков, но и создавать свои собственные. Обобщенными могут быть классы, интерфейсы и записи. Также поддерживается создание обобщенных методов (процедур и функций).

4.1. Обобщенные методы

Самым простым примером обобщенного метода может служить процедура для обмена значений переменных:

Листинг 7 - Пример дженериковой процедуры `Swap<T>`

```
01. ...
02.
03. TSwapper = class
04.     class procedure Swap<T>(var a, b: T);
05. end;
06.
07. class procedure TSwapper.Swap<T>(var a, b: T);
08. var
09.     temp: T;
10. begin
11.     temp := b;
12.     b := a;
13.     a := temp;
14. end;
15.
16. ...
```

Использовать такую процедуру можно следующим образом:

Листинг 8 - Использование дженериковой процедуры Swap<T>

```
01. ...
02.
03.   TPoint<T> = record
04.     X: T;
05.     Y: T;
06.     // Добавим следующую маленькую функцию,
07.     // чтобы легче производить инициализацию записи
08.     class function Create(X, Y: T): TPoint<T>; static;
09.   end;
10.
11. class function TPoint<T>.Create(X, Y: T): TPoint<T>;
12. begin
13.   Result.X := X;
14.   Result.Y := Y;
15. end;
16.
17. var
18.   a, b: Integer;
19.   s1, s2: string;
20.   p1, p2: TPoint<Double>;
21. begin
22.   // Теперь с помощью обобщенной процедуры
23.   // мы имеем возможность обменивать значения целых чисел...
24.   a := 1;
25.   b := 10;
26.   Writeln(Format('До обмена: a=%d; b=%d', [a, b]));
27.   TSwapper.Swap<Integer>(a, b);
28.   Writeln(Format('После обмена: a=%d; b=%d', [a, b]));
29.   Writeln;
30.
31.   // ... строк...
32.   s1 := 'Delphi';
33.   s2 := 'XE';
34.   Writeln(Format('До обмена: s1=%s; s2=%s', [s1, s2]));
35.   TSwapper.Swap<string>(s1, s2);
36.   Writeln(Format('После обмена: s1=%s; s2=%s', [s1, s2]));
37.   Writeln;
38.
39.   // ... и даже нашего типа TPoint<T>!
40.   p1 := TPoint<Double>.Create(5.5, -10);
41.   p2 := TPoint<Double>.Create(-1.5, 10);
42.   Writeln(Format('До обмена: p1: (%.2f, %.2f); p2: (%.2f, %.2f);', [p1.X, p1.Y, p2.X,
43.     p2.Y]));
44.   TSwapper.Swap<TPoint<Double>>(p1, p2);
45.   Writeln(Format('После обмена: p1: (%.2f, %.2f); p2: (%.2f, %.2f);', [p1.X, p1.Y,
46.     p2.X, p2.Y]));
47. end;
```

Результат приведен на Рисунке 1:

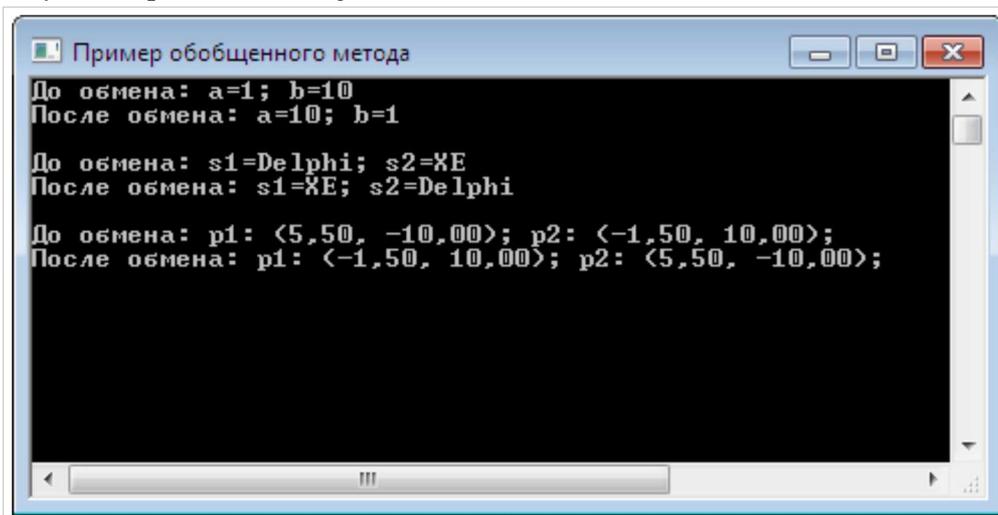


Рисунок 1 - Пример использования Swap<T>

4.2. Обобщенные классы

Приведем пример обобщенного класса массива:

Листинг 9 - Пример обобщенного класса массива *TGenericArray<T>*

```
01. ...
02.
03. type
04. // Обобщенный класс - одномерный массив
05. TGenericArray<T> = class
06. private
07. // Обобщенные члены класса
08. FItems: array of T;
09. function GetItem(Index: Integer): T;
10. function GetCount: Integer;
11. procedure SetItem(Index: Integer; Value: T);
12. public
13. // Обобщенный конструктор
14. constructor Create;
15. // Обобщенные методы
16. function IndexOf(Value: T): Integer;
17. function SetSize(NewSize: Integer): Boolean;
18. // Обобщенные свойства
19. property Items[I: Integer]: T read GetItem write SetItem; default;
20. property Count: Integer read GetCount;
21. end;
22.
23.
24. { TGenericArray<T> }
25.
26. constructor TGenericArray<T>.Create;
27. begin
28.   SetSize(0);
29. end;
30.
31. function TGenericArray<T>.GetCount: Integer;
32. begin
33.   Result := Length(FItems);
34. end;
35.
36. function TGenericArray<T>.GetItem(Index: Integer): T;
37. begin
38.   if (Index < 0) or (Index >= Length(FItems)) then
39.     raise EArgumentOutOfRangeException.Create(SArgumentOutOfRangeException);
40.   Result := FItems[Index];
41. end;
42.
43. function TGenericArray<T>.IndexOf(Value: T): Integer;
44. begin
45.   // здесь может быть поиск элемента
46. end;
47.
48. procedure TGenericArray<T>.SetItem(Index: Integer; Value: T);
49. begin
50.   if (Index < 0) or (Index >= Length(FItems)) then
51.     raise EArgumentOutOfRangeException.Create(SArgumentOutOfRangeException);
52.   FItems[Index] := Value;
53. end;
54.
55. function TGenericArray<T>.SetSize(NewSize: Integer): Boolean;
56. begin
57.   Result := NewSize > 0;
58.   if Result then
59.     SetLength(FItems, NewSize);
60. end;
61.
62. ...
```

Посмотрим на вариант его использования:

```
01. ...
02.
03. var
04.   IntArray: TGenericArray<Integer>;
05.   DoubleArray: TGenericArray<Double>;
06.   i: Integer;
07.
08. begin
09.   // Создаем "целочисленный вариант" нашего класса-массива
10.   IntArray := TGenericArray<Integer>.Create;
11.   IntArray.SetSize(3);
12.   IntArray[0] := 1;
13.   IntArray[1] := 2;
14.   IntArray[2] := 3;
15.
16.   // а теперь "дробный"
17.   DoubleArray := TGenericArray<Double>.Create;
18.   DoubleArray.SetSize(4);
19.   DoubleArray[0] := 5;
20.   DoubleArray[1] := 2.5;
21.   DoubleArray[2] := -3;
22.   DoubleArray[3] := 3;
23.
24.   Writeln(Format('В IntArray %d элем.:', [IntArray.Count]));
25.   for i := 0 to IntArray.Count - 1 do
26.     Writeln(Format('%d-й элемент = %d', [i, IntArray[i]]));
27.
28.   Writeln;
29.
30.   Writeln(Format('В DoubleArray %d элем.:', [DoubleArray.Count]));
31.   for i := 0 to DoubleArray.Count - 1 do
32.     Writeln(Format('%d-й элемент = %.1f', [i, DoubleArray[i]]));
33.
34.   FreeAndNil(IntArray);
35.   FreeAndNil(DoubleArray);
36.
37. ...
```

Результат приведен на *Рисунке 2*.

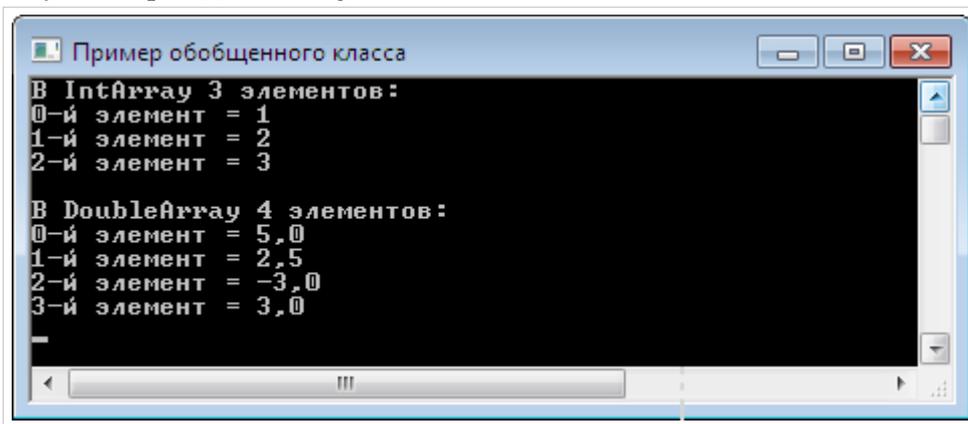


Рисунок 2 - Пример использования `TGenericArray<T>`

4.3. Обобщенные записи

Пример обобщенной записи `TPoint<T>` уже был приведен в начале раздела. Гляньте на нее еще разок.

5. Заключение

Мы познакомились с синтаксисом дженериков, их преимуществами и возможностями в Delphi. В следующем разделе мы рассмотрим системные обобщенные классы из модулей `Generics.Defaults` и `Generics.Collections`.

Часть 2 - Системные классы

1. Введение

Рассмотрим стандартные обобщенные классы в Delphi и их функциональные особенности.

Большинство информации будет представлено в виде демо-примеров, которые при желании можно будет легко воспроизвести у себя на ПК.

2. Generics.Defaults

2.1. TComparer<T>

TComparer является базовым обобщенным классом, реализует интерфейс IComparer и предназначен для создания компараторов - классов, отвечающих за сравнение других классов или типов.

Создавать компараторы для некоторых стандартных типов (о которых можно получить информацию через RTTI) очень просто: для этого существует классовая функция TComparer.Default. Так, компаратор для целых чисел или строк создается через TComparer<Integer>.Default и TComparer<string>.Default соответственно. А как же сравнивать нестандартные типы или собственные классы? Для этого потребуется создать собственный компаратор на основе TComparer<T> и всего лишь переопределить функцию Compare:

Листинг 11 - Создание пользовательского компаратора для класса TCustomer

```
01. uses
02.   SysUtils, Generics.Defaults;
03.
04. ...
05.
06. // Создадим компаратор для класса TCustomer из Листинга 4
07.
08. // Немного расширим класс, пусть у клиента будет банковский счет
09. TCustomer = class
10. private
11.   FName: string;
12.   FMoney: Currency;
13. private
14.   constructor Create(const Name: string; Money: Currency);
15.   property Name: string read FName write FName;
16.   property Money: Currency read FMoney write FMoney;
17. end;
18.
19. // Объявляем наш компаратор и переопределяем функцию Compare
20. TCustomerComparer = class(TComparer<TCustomer>)
21.   function Compare(const Left, Right: TCustomer): Integer; override;
22. end;
23.
24. function TCustomerComparer.Compare(const Left, Right: TCustomer): Integer;
25. begin
26.   // Переопределяемый метод Compare должен возвращать:
27.   // 0 - при равенстве значений
28.   // >0 если первый параметр больше второго (Left > Right)
29.   // <0 в остальных случаях (Left < Right)
30.
31.   // Будем сортировать клиентов сначала по имени, а затем по сумме счета
32.   Result := CompareStr(Left.Name, Right.Name);
33.   if Result = 0 then
34.     Result := Left.Money - Right.Money;
35. end;
36.
37. ...
```

Кроме того, можно и не создавать класс, а использовать классовую функцию - TComparer<T>.Construct в том случае, когда компаратор передается как параметр метода. Сделать это необходимо через анонимную функцию того же вида, что и рассмотренная выше Compare:

Листинг 12 - "Конструирование" пользовательского компаратора с использованием функции Construct и анонимного метода

```
01. ...
02.
03. // Предположим, что метод Sort в TCustomersManager принимает компаратор
04. // в качестве параметра для сортировки имеющихся клиентов
05. TCustomersManager.Sort(TComparer<TCustomer>.Construct(
06.     function (const Left, Right: Integer): Integer
07.     begin
08.         Result := CompareStr(Left.Name, Right.Name);
09.         if Result = 0 then
10.             Result := Left.Money - Right.Money;
11.         end));
12.
13. ...
```

Где и как использовать компараторы в различных вариациях, мы увидим уже в следующих разделах.

3. Generics.Collections

3.1. TArray

Создавать экземпляр этого класса не нужно. TArray содержит 2 статических метода:

- `Sort`. Использует алгоритм быстрой сортировки, может принимать компаратор в качестве параметра
- `BinarySearch`. Ищет элемент в массиве и возвращает True если находит. Также принимает компаратор, требует, чтобы массив был предварительно отсортирован

3.1.1. Сортировка и поиск элементов в одномерном массиве

Рассмотрим возможности сортировки и поиска на примере простого одномерного массива.

Листинг 13 - Сортировка и поиск элементов одномерного целочисленного массива с использованием TArray

```
01. ...
02.
03. uses
04.     SysUtils, Types,
05.     Generics.Collections, Generics.Defaults;
06.
07. function CompareIntReverse(const Left, Right: Integer): Integer;
08. begin
09.     // Сравниваем элементы "наоборот" и получаем обратный порядок
10.     Result := Right - Left;
11. end;
12.
13. procedure PrintMatrix(A: TIntegerDynArray);
14. var
15.     item: Integer;
16. begin
17.     for item in A do
18.         Write(item, ' ');
19.     Writeln; Writeln;
20. end;
21.
22. var
23.     A: TIntegerDynArray;
24.     FoundIndex: Integer;
25.
26. begin
27.
28.     ...
29.
30.     SetLength(A, 5);
31.     A[0] := 1;
```

```

32. A[1] := 6;
33. A[2] := 3;
34. A[3] := 2;
35. A[4] := 9;
36.
37. // Распечатаем, что есть
38. Writeln('Исходный массив:');
39. PrintMatrix(A);
40.
41. // Сортируем по возрастанию без компаратора
42. TArray.Sort<Integer>(A);
43. Writeln('По возрастанию Sort без параметров:');
44. PrintMatrix(A);
45.
46. // Сортируем по убыванию, конструируя компаратор
47. // с помощью анонимного метода
48. TArray.Sort<Integer>(A, TComparer<Integer>.Construct(
49.     function (const Left, Right: Integer): Integer
50.     begin
51.         Result := Right - Left;
52.     end));
53. Writeln('По убыванию с TComparer<Integer>.Construct(анонимный метод:');
54. PrintMatrix(A);
55.
56. // Опять сортируем по возрастанию с применением компаратора по умолчанию
57. TArray.Sort<Integer>(A, TComparer<Integer>.Default);
58. Writeln('По возрастанию с TComparer<Integer>.Default:');
59. PrintMatrix(A);
60.
61. // И снова по убыванию с использованием собственного компаратора
62. TArray.Sort<Integer>(A, TComparer<Integer>.Construct(CompareIntReverse));
63. Writeln('По убыванию TComparer<Integer>.Construct(CompareIntReverse:');
64. PrintMatrix(A);
65.
66. // Выполняем поиск несуществующего элемента
67. Writeln('BinarySearch несуществующего элемента');
68. if TArray.BinarySearch<Integer>(A, 5, FoundIndex) then
69.     Writeln('5-ка найдена, ее индекс ', FoundIndex)
70. else
71.     Writeln('5-ки в массиве нет!');
72. Writeln;
73.
74. // Выполняем поиск существующего элемента
75. Writeln('BinarySearch существующего элемента');
76. if TArray.BinarySearch<Integer>(A, 6, FoundIndex) then
77.     Writeln('6-ка найдена, ее индекс ', FoundIndex)
78. else
79.     Writeln('6-ки в массиве нет!');
80. Writeln;
81.
82. // Выполняем поиск с нашим компаратором CompareIntReverse
83. Writeln('BinarySearch существующего элемента с компаратором');
84. if TArray.BinarySearch<Integer>(A, 6, FoundIndex,
85.     TComparer<Integer>.Construct(CompareIntReverse)) then
86.     Writeln('6-ка найдена, ее индекс ', FoundIndex)
87. else
88.     Writeln('6-ки в массиве нет!');
89. Writeln;
90.
91. FreeAndNil(A);
92.
93.
94. ...

```

Результат работы приведен на *Рисунке 3*:

```

Работа с TArray - одномерный целочисленный массив
Исходный массив:
1 6 3 2 9
По возрастанию Sort без параметров:
1 2 3 6 9
По убыванию с TComparer<Integer>.Construct(анонимный метод):
9 6 3 2 1
По возрастанию с TComparer<Integer>.Default:
1 2 3 6 9
По убыванию TComparer<Integer>.Construct(CompareIntReverse):
9 6 3 2 1
BinarySearch несуществующего элемента
5-ки в массиве нет!
BinarySearch существующего элемента
6-ки в массиве нет!
BinarySearch существующего элемента с компаратором
6-ка найдена, ее индекс 1

```

Рисунок 3 - Результаты поиска и сортировки с использованием TArray.Sort и TArray.BinarySearch

3.1.2. Сортировка двумерного массива

Посмотрим, как осуществить сортировку двумерного массива:

Листинг 14 - Сортировка элементов двумерного целочисленного массива с использованием TArray

```

01. ...
02.
03. uses
04.   SysUtils, Math, Types,
05.   Generics.Collections,
06.   Generics.Defaults;
07.
08. type
09.   TDoubleIntegerArray = array of TIntegerDynArray;
10.
11. procedure PrintMatrix(A: TDoubleIntegerArray);
12. var
13.   i, j: Integer;
14. begin
15.   for i := Low(A) to High(A) do
16.     begin
17.       for j := Low(A[0]) to High(A[0]) do
18.         Write(A[i, j]: 3, ' ');
19.       Writeln;
20.     end;
21.   Writeln; Writeln;
22. end;
23.
24. var
25.   A: TDoubleIntegerArray;
26.   FoundIndex: Integer;
27.   i, j: Integer;
28.
29. begin
30.
31. ...
32.
33. // Заполним целочисленный массив случайными числами [1..50]
34. SetLength(A, 4, 7);
35. Randomize;
36. for i := Low(A) to High(A) do
37.   for j := Low(A[0]) to High(A[0]) do
38.     A[i, j] := Math.RandomRange(1, 50);
39.
40.

```

```

41. // Приравниваем часть элементов для дальнейшей "каскадной" сортировки
42. A[1, 0] := A[0, 0];
43. A[2, 0] := A[0, 0];
44. A[1, 1] := A[0, 1];
45.
46. // Распечатаем, что получилось
47. Writeln('Исходный массив:');
48. PrintMatrix(A);
49.
50. // Сортируем по убыванию по 1-й колонке, конструируя компаратор
51. // с помощью анонимного метода
52. TArray.Sort<TIntegerDynArray>(A, TComparer<TIntegerDynArray>.Construct(
53.     function (const Left, Right: TIntegerDynArray): Integer
54.     begin
55.         Result := Right[0] - Left[0];
56.     end));
57. Writeln('По убыванию в 1 столбце:');
58. PrintMatrix(A);
59.
60. // Сортируем по убыванию по 1-й колонке "каскадом"
61. TArray.Sort<TIntegerDynArray>(A, TComparer<TIntegerDynArray>.Construct(
62.     function (const Left, Right: TIntegerDynArray): Integer
63.     var
64.         i: Integer;
65.     begin
66.         i := 0;
67.         repeat
68.             Result := Right[i] - Left[i];
69.             Inc(i);
70.         until ((Result <> 0) or (i = Length(Left)));
71.     end));
72. Writeln('Каскадная сортировка, начиная с 1-го столбца:');
73. PrintMatrix(A);
74.
75. ...
76.

```

Результат работы приведен на *Рисунке 4*:

```

Работа с TArray - двумерный целочисленный массив
Исходный массив :
9 48 47 27 1 46 38
9 48 47 33 46 7 6
9 42 8 3 30 46 34
16 14 19 45 37 44 18

По убыванию в 1 столбце:
16 14 19 45 37 44 18
9 42 8 3 30 46 34
9 48 47 27 1 46 38
9 48 47 33 46 7 6

Каскадная сортировка, начиная с 1-го столбца:
16 14 19 45 37 44 18
9 48 47 33 46 7 6
9 48 47 27 1 46 38
9 42 8 3 30 46 34

```

Рисунок 4 - Результаты сортировки двумерного массива с использованием `TArray.Sort`

3.2. TDictionary<T> и TObjectDictionary<T>

Данные классы представляют из себя коллекцию пар ключ-значение, или попросту словарь. Рассмотрим работу с этими классами на примере TDictionary<T>.

3.2.1. Работа со словарем (на примере TDictionary<T>)

Основные возможности и свойства TDictionary<T> рассмотрим на следующем примере:

Листинг 15 - Использование TDictionary<T> в качестве коллекции данных для телефонной книги

```
01. ...
02.
03. uses
04.   SysUtils, Generics.Collections, Generics.Defaults;
05.
06. type
07.
08.   // Информация о владельце номера
09.   TSubscriberInfo = record
10.     Name, SName: string;
11.     class function Create(const Name, SName: string): TSubscriberInfo; static;
12.     function ToString: string;
13.   end;
14.
15. class function TSubscriberInfo.Create(const Name,
16.   SName: string): TSubscriberInfo;
17. begin
18.   Result.Name := Name;
19.   Result.SName := SName;
20. end;
21.
22. function TSubscriberInfo.ToString: string;
23. begin
24.   Result := Format('%s %s', [Name, SName]);
25. end;
26.
27. var
28.   // Объявляем "словарь"
29.   // ключом будет номер телефона, по которому можно будет
30.   // определить информацию о владельце в виде TSubscriberInfo
31.   TelephoneDirectory: TDictionary<string, TSubscriberInfo>;
32.   tempInfo: TSubscriberInfo;
33. begin
34.
35.   ...
36.
37.   // Создаем справочник
38.   // Конструктор имеет несколько презагруженных вариантов, позволяющих
39.   // установить емкость контейнера, компаратор для значений или
40.   // первоначальный данные - мы используем самый простой вариант
41.   TelephoneDirectory := TDictionary<string, TSubscriberInfo>.Create;
42.
43.   // -----
44.   // 1) Добавление в словарь
45.
46.   // Добавляем абонентов в справочник
47.   TelephoneDirectory.Add('9101111111', TSubscriberInfo.Create('Арнольд',
48.     'Шварценеггер'));
49.   TelephoneDirectory.Add('9102222222', TSubscriberInfo.Create('Джессика', 'Альба'));
50.   TelephoneDirectory.Add('9103333333', TSubscriberInfo.Create('Бред', 'Питт'));
51.   TelephoneDirectory.Add('9104444444', TSubscriberInfo.Create('Бред', 'Питт'));
52.   TelephoneDirectory.Add('9105555555', TSubscriberInfo.Create('Сандра', 'Баллок'));
53.   // Добавляем нового абонента и заменяем, если такой номер уже есть
54.   TelephoneDirectory.AddOrSetValue('9104444444',
55.     TSubscriberInfo.Create('Анджелина', 'Джоли'));
56.
57.   // -----
58.   // 2) Получение, поиск, работа с элементами
59.
60.   // Имеется ли номер телефона (ключ) - ContainsKey
61.   if TelephoneDirectory.ContainsKey('9105555555') then
```

```

61.     Writeln('Номер 9105555555 зарегистрирован!');
62.     // Имеется ли абонент (значение) - ContainsValue
63.     tempInfo := TSubscriberInfo.Create('Сандра', 'Баллок');
64.     if TelephoneDirectory.ContainsValue(tempInfo) then
65.         Writeln(Format('%s есть в справочнике!', [tempInfo.ToString]));
66.     // Пробуем получить информацию по телефону через TryGetValue
67.     if TelephoneDirectory.TryGetValue('9104444444', tempInfo) then
68.         Writeln(Format('Номер 9104444444 у абонента %s', [tempInfo.ToString]));
69.     // Обращение по номеру телефона
70.     Writeln(Format('Абонент с номером 9101111111: %s',
71.         [TelephoneDirectory['9101111111'].ToString]));
72.     // Кол-во людей в справочнике
73.     Writeln(Format('Всего абонентов в справочнике: %d', [TelephoneDirectory.Count]));
74.     // -----
75.     // 3) Удаление элементов
76.
77.     // Шварценеггера сейчас не будет в списке =)
78.     TelephoneDirectory.Remove('9101111111');
79.     // Полностью очищаем список
80.     TelephoneDirectory.Clear;
81.
82.     FreeAndNil(TelephoneDirectory);
83.
84.     Readln;
85. end.

```

3.2.2. События OnKeyNotify и OnValueNotify

На случай, если Вы хотите отслеживать изменения в словаре, предусмотрены события OnKeyNotify и OnValueNotify, срабатывающие при добавлении/удалении ключа и значения соответственно. Добавим небольшой фрагмент кода к предыдущему Листингу и продемонстрируем их работу:

Листинг 16 - Использование событий TDictionary<T>

```

01. ...
02.
03. uses
04.     SysUtils, Generics.Collections, Generics.Defaults;
05.
06. type
07.
08. ...
09.
10.     // Класс, содержащий обработчики добавления/удаления элементов словаря
11.     TDictionaryEventsHandler = class
12.     public
13.         // Синтаксис процедур един, единственное, в обоих случаях для Item необходимо
14.         // указать тип согласно типам нашего ключа и значения
15.         class procedure OnKeyNotify(Sender: TObject; const Item: string;
16.             Action: TCollectionNotification);
17.         class procedure OnValueNotify(Sender: TObject; const Item: TSubscriberInfo;
18.             Action: TCollectionNotification);
19.     end;
20.
21. class procedure TDictionaryEventsHandler.OnKeyNotify(Sender: TObject;
22.     const Item: string; Action: TCollectionNotification);
23. begin
24.     case Action of
25.     cnAdded:
26.         Writeln(Format('OnKeyNotify! Номер %s добавлен!', [Item]));
27.     cnRemoved:
28.         Writeln(Format('OnKeyNotify! Номер %s удален!', [Item]));
29.     end;
30. end;
31.
32. class procedure TDictionaryEventsHandler.OnValueNotify(Sender: TObject;
33.     const Item: TSubscriberInfo; Action: TCollectionNotification);
34. begin
35.     case Action of
36.     cnAdded:
37.         Writeln(Format('OnValueNotify! Абонент %s добавлен!', [Item.ToString]));

```

```

38.     cnRemoved:
39.         Writeln(Format('OnValueNotify! Абонент %s удален!', [Item.ToString]));
40.     end;
41. end;
42.
43. ...
44.
45. begin
46.     ...
47.
48.     // -----
49.     // 4) События добавления/удаления значений
50.
51.     // События OnKeyNotify и OnValueNotify предназначены для "слежения"
52.     // за добавлением/удалением ключей и значений соответственно
53.     // Эти обработчики мы реализовали выше в классе TDictionaryEventsHandler
54.     TelephoneDirectory.OnKeyNotify := TDictionaryEventsHandler.OnKeyNotify;
55.     TelephoneDirectory.OnValueNotify := TDictionaryEventsHandler.OnValueNotify;
56.
57.     Writeln;
58.     // Пробуем, как откликаются события
59.     TelephoneDirectory.Add('9101111111', TSubscriberInfo.Create('Арнольд',
60.         'Шварценеггер'));
61.     TelephoneDirectory.Add('9102222222', TSubscriberInfo.Create('Джессика', 'Альба'));
62.     TelephoneDirectory.Clear;
63.
64.     ...
65. end.

```

Результат работы приведен на *Рисунке 5*:

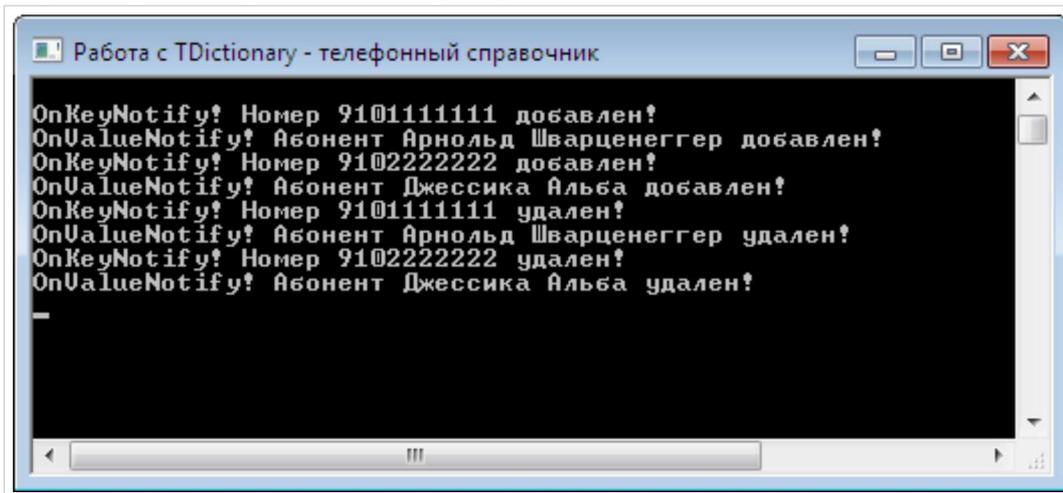


Рисунок 5 - Обработка событий OnKeyNotify и OnValueNotify в TDictionary<T>

3.2.3. Ключи, значения и TArray

Получить доступ к элементам словаря можно сделать несколькими способами:

Листинг 17 - Доступ к элементам TDictionary<T>

```

01. ...
02. uses
03.     SysUtils, Generics.Collections;
04.
05. ...
06. var
07.     TelephoneDirectory: TDictionary<string, TSubscriberInfo>;
08.     // TPair - обобщенная запись, предназначенная для хранения
09.     // пары ключ-значение и не обладающая какой-либо другой функциональностью,
10.     // поэтому я не освещаю ее отдельно
11.     TTelephoneArray: TArray<TPair<string, TSubscriberInfo>>;
12.     TTelephoneArrayItem: TPair<string, TSubscriberInfo>;
13.     PhoneNumber: string;
14.     Subscriber: TSubscriberInfo;
15. begin

```

```

16. ...
17. TelephoneDirectory := TDictionary<string, TSubscriberInfo>.Create;
18.
19. TelephoneDirectory.Add('9101111111', TSubscriberInfo.Create('Моника', 'Белуччи'));
20. TelephoneDirectory.Add('9102222222', TSubscriberInfo.Create('Сильвестр',
    'Сталлоне'));
21. TelephoneDirectory.Add('9103333333', TSubscriberInfo.Create('Брюс', 'Уиллис'));
22.
23. // Показываем ключи (телефоны)
24. Writeln('Ключи (телефоны):');
25. for PhoneNumber in TelephoneDirectory.Keys do
26.     Writeln(PhoneNumber);
27. Writeln;
28. // Показываем значения (абонентов)
29. Writeln('Значения (абоненты):');
30. for Subscriber in TelephoneDirectory.Values do
31.     Writeln(Subscriber.ToString);
32. Writeln;
33. // Теперь все вместе
34. Writeln('Список абонентов с телефонами:');
35. for PhoneNumber in TelephoneDirectory.Keys do
36.     Writeln(Format('%s: %s',
37.         [PhoneNumber, TelephoneDirectory[PhoneNumber].ToString]));
38. Writeln;
39.
40. // Кроме того, мы можем "экспортировать" словарь в знакомый нам TArray
41. // Отсортируем полученный массив и выведем на экран
42. TTelephoneArray := TelephoneDirectory.ToArray;
43. TArray.Sort<TPair<string, TSubscriberInfo>>(
44.     TTelephoneArray, TComparer<TPair<string, TSubscriberInfo>>.Construct(
45.         function (const Left, Right: TPair<string, TSubscriberInfo>): Integer
46.         begin
47.             // Сравним сначала полные имена, а затем телефоны при необходимости
48.             Result := CompareStr(Left.Value.ToString, Right.Value.ToString);
49.             if Result = 0 then
50.                 Result := CompareStr(Left.Key, Right.Key);
51.         end));
52. // Печатаем
53. Writeln('Отсортированный через TArray список абонентов с телефонами:');
54. for TTelephoneArrayItem in TTelephoneArray do
55.     Writeln(Format('%s: %s',
56.         [TTelephoneArrayItem.Value.ToString, TTelephoneArrayItem.Key]));
57.
58. FreeAndNil(TelephoneDirectory);
59. ...

```

Результат работы приведен на *Рисунке 6*:

Рисунок 6 - Доступ к ключам и значениям *TDictionary<T>* и экспорт в *TArray*

3.3. TList<T> и TObjectList<T>

TList<T> и TObjectList<T> - упорядоченные списки.

3.3.1. Работа со списком (на примере TObjectList<T>)

Стандартные возможности по работе с классом TObjectList<T> приведены в Листинге 18:

Листинг 18 - Использование TObjectList<T> в качестве футбольного "менеджера"

```
001. ...
002.
003. uses
004.     SysUtils, DateUtils,
005.     Generics.Collections, Generics.Defaults;
006.
007. type
008.     TPlayer = class
009.     public
010.         Name, Team: string;
011.         BirthDay: TDateTime;
012.         NTeamGoals: Byte; // Кол-во голов за национальную сборную
013.         constructor Create(const Name: string; BirthDay: TDateTime;
014.             const Team: string; NTeamGoals: Byte = 0);
015.         function ToString: string;
016.     end;
017.
018. constructor TPlayer.Create(const Name: string; BirthDay: TDateTime;
019.     const Team: string; NTeamGoals: Byte);
020. begin
021.     Self.Name := Name;
022.     Self.BirthDay := BirthDay;
023.     Self.Team := Team;
024.     Self.NTeamGoals := NTeamGoals;
025. end;
026.
027. function TPlayer.ToString: string;
028. begin
029.     Result := Format('%s - Возраст: %d Сборная: %s Голов: %d',
030.         [Name, DateUtils.YearsBetween(Date, BirthDay),
031.         Team, NTeamGoals]);
032. end;
033.
034. var
035.     // Объявляем TObjectList для хранения TPlayer
036.     PlayersList: TObjectList<TPlayer>;
037.     Player: TPlayer;
038.
039. begin
040.     ...
041.
042.     PlayersList := TObjectList<TPlayer>.Create;
043.
044.     // -----
045.     // 1) Добавление элементов
046.
047.     // "Простое" добавление в конец списка
048.     PlayersList.Add(
049.         TPlayer.Create('Роналдо', EncodeDate(1976, 09, 22), 'Бразилия', 62));
050.     PlayersList.Add(
051.         TPlayer.Create('Зинедин Зидан', EncodeDate(1972, 06, 23), 'Франция', 31));
052.     WriteLn(Format('Индекс добавляемого игрока: %d',
053.         [PlayersList.Add(TPlayer.Create('Юрген Клинсманн',
054.             EncodeDate(1964, 07, 30),
055.             'Германия', 47))]));
056.     // Добавляем в указанную позицию
057.     PlayersList.Insert(0,
058.         TPlayer.Create('Луиш Фигу', EncodeDate(1972, 11, 4), 'Португалия', 33));
059.     // Добавляем несколько футболистов через InsertRange (AddRange работает аналогично)
060.     PlayersList.InsertRange(0,
061.         [TPlayer.Create('Девид Бекхэм', EncodeDate(1975, 05, 2), 'Англия', 17),
062.         TPlayer.Create('Алессандро Дель Пьеро', EncodeDate(1974, 11, 9), 'Италия', 27),
```

```

063.     TPlayer.Create('Рауль', EncodeDate(1977, 06, 27), 'Испания', 44));
064.     // Добавляем заранее созданный экземпляр класса
065.     Player := TPlayer.Create('Рауль', EncodeDate(1977, 06, 27), 'Испания', 44);
066.     PlayersList.Add(Player);
067.
068.
069.     // -----
070.     // 2) Доступ и проверка наличия элементов
071.
072.     // Имеется ли игрок в списке - Contains
073.     if PlayersList.Contains(Player) then
074.         Writeln('Рауль есть в списке!');
075.     // Индекс игрока и кол-во элементов в списке
076.     Writeln(Format('Рауль %d-й в списке из %d футболистов.',
077.         [PlayersList.IndexOf(Player) + 1, PlayersList.Count]));
078.     // Доступ по индексу
079.     Writeln(Format('1-й в списке: %s', [PlayersList[0].ToString]));
080.     // "Переворачиваем" элементы
081.     PlayersList.Reverse;
082.     Writeln('Элементы списка были "перевернуты"');
083.
084.
085.     // -----
086.     // 3) Перемещение и удаление элементов
087.
088.     // Меняем игроков в списке местами
089.     PlayersList.Exchange(0, 1);
090.     // Перемещаем 1 игрока обратно
091.     PlayersList.Move(1, 0);
092.
093.     // Удаляем элемент по индексу
094.     PlayersList.Delete(5);
095.     // Или несколько элементов (2), начиная с индекса (5)
096.     PlayersList.DeleteRange(5, 2);
097.     // Remove удаляет элемент из списка, если элемент существует
098.     // вернется его индекс в списке, иначе -1
099.     Writeln(Format('Удален %d-й игрок', [PlayersList.Remove(Player) + 1]));
100.
101.     // Extract возвращает элемент, удаляя его из списка
102.     // В нашем случае Player будет = nil, т.к. Рауля мы уже удалили через Remove
103.     Player := PlayersList.Extract(Player);
104.     if Assigned(Player) then
105.         Writeln(Format('Извлечен: %s', [Player.ToString]));
106.
107.     // Очищаем список полностью
108.     PlayersList.Clear;
109.
110.     FreeAndNil(PlayersList);
111.
112.     Readln;
113. end.

```

3.3.2. Поиск и сортировка

Методы сортировки и поиска полностью идентичны соответствующим методам класса TArray:

Листинг 19 - Сортировка и поиск в TObjectList<T>

```

01. ...
02.
03. // Функция сортировки по убыванию голов за сборную
04. function ComparePlayersByGoalsDecs(const Player1, Player2: TPlayer): Integer;
05. begin
06.     Result := Player2.NTeamGoals - Player1.NTeamGoals;
07. end;
08.
09. var
10. ...
11.     FoundIndex: Integer;
12.
13. begin
14.
15. ...

```

```

16.
17.   PlayersList := TObjectList<TPlayer>.Create;
18.
19.   PlayersList.Add(
20.     TPlayer.Create('Зинедин Зидан', EncodeDate(1972, 06, 23), 'Франция', 31));
21.   PlayersList.Add(
22.     TPlayer.Create('Роналдо', EncodeDate(1976, 09, 22), 'Бразилия', 62));
23.   PlayersList.Add(
24.     TPlayer.Create('Юрген Клинсманн', EncodeDate(1964, 07, 30), 'Германия', 47));
25.
26.   // Сортируем с использованием ComparePlayersByGoalsDecs
27.   PlayersList.Sort(TComparer<TPlayer>.Construct(ComparePlayersByGoalsDecs));
28.   Writeln('Список игроков:');
29.   for Player in PlayersList do
30.     Writeln(Player.ToString);
31.   Writeln;
32.
33.   // Найдем Роналдо в списке
34.   // Как и в случае с TArray BinarySearch требует, чтобы список был отсортирован
35.   // Нюанс - по сути, выполняющий тоже самое, что и IndexOf
36.   // BinarySearch обычно быстрее (первый не требует сортировки списка)
37.   Player := PlayersList[0];
38.   if PlayersList.BinarySearch(Player, FoundIndex,
39.     TComparer<TPlayer>.Construct(ComparePlayersByGoalsDecs)) then
40.     Writeln(Format('Роналдо идет отсортированном списке под №%d', [FoundIndex + 1]));
41.
42.   ...

```

Результат работы приведен на *Рисунке 7*:

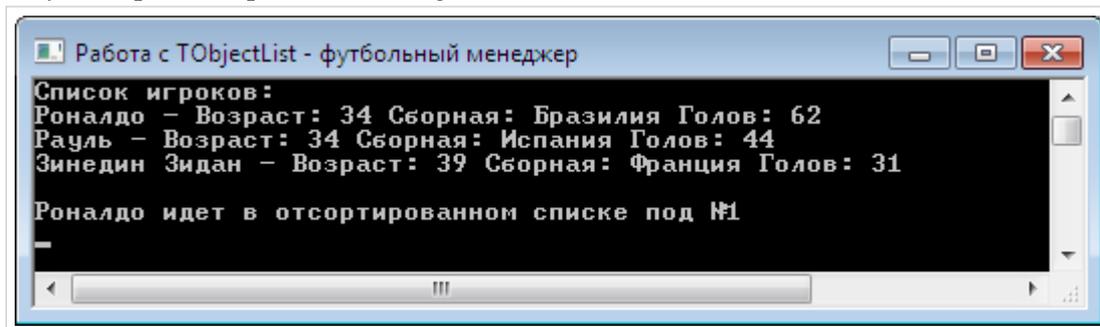


Рисунок 7 - Сортировка и поиск в TObjectList<T>

Кроме того TObjectList<T> имеет функцию ToArray, которая работает так же, как и в TDictionary<T>, поэтому не будем приводить ее еще раз.

3.3.3. Событие OnNotify

Событие OnNotify возникает при изменении содержимого списка. Приведем пример ее использования:

Листинг 20 - Событие OnNotify в TObjectList<T>

```

01. // Класс, содержащий обработчики добавления/удаления элементов списка
02. TListEventsHandler = class
03. public
04.   class procedure OnListChanged(Sender: TObject; const Item: TPlayer;
05.     Action: TCollectionNotification);
06. end;
07.
08. class procedure TListEventsHandler.OnListChanged(Sender: TObject; const Item:
09.   TPlayer;
10.   Action: TCollectionNotification);
11. var
12.   Mes: string;
13. begin
14.   // В отличие от TDictionary у нас добавляется Action = cnExtracted
15.   case Action of
16.     cnAdded:
17.       Mes := 'добавлен в список!';
18.     cnRemoved:
19.       Mes := 'удален из списка!';
20.     cnExtracted:

```

```

20.     Mes := 'извлечен из списка!';
21. end;
22. Writeln(Format('Футболист %s %s ', [Item.ToString, Mes]));
23. end;
24.
25. ...
26.
27. begin
28.     ...
29.
30.     PlayersList := TObjectList<TPlayer>.Create;
31.     PlayersList.OnNotify := TListEventsHandler.OnListChanged;
32.
33.     PlayersList.Add(
34.         TPlayer.Create('Зинедин Зидан', EncodeDate(1972, 06, 23), 'Франция', 31));
35.     PlayersList.Add(
36.         TPlayer.Create('Рауль', EncodeDate(1977, 06, 27), 'Испания', 44));
37.     PlayersList.Add(
38.         TPlayer.Create('Роналдо', EncodeDate(1976, 09, 22), 'Бразилия', 62));
39.
40.     PlayersList.Delete(1);
41.     Player := PlayersList.Extract(PlayersList[0]);
42.     PlayersList.Clear;
43.
44.     FreeAndNil(PlayersList);
45.
46. ...

```

Результат работы приведен на *Рисунке 8*:

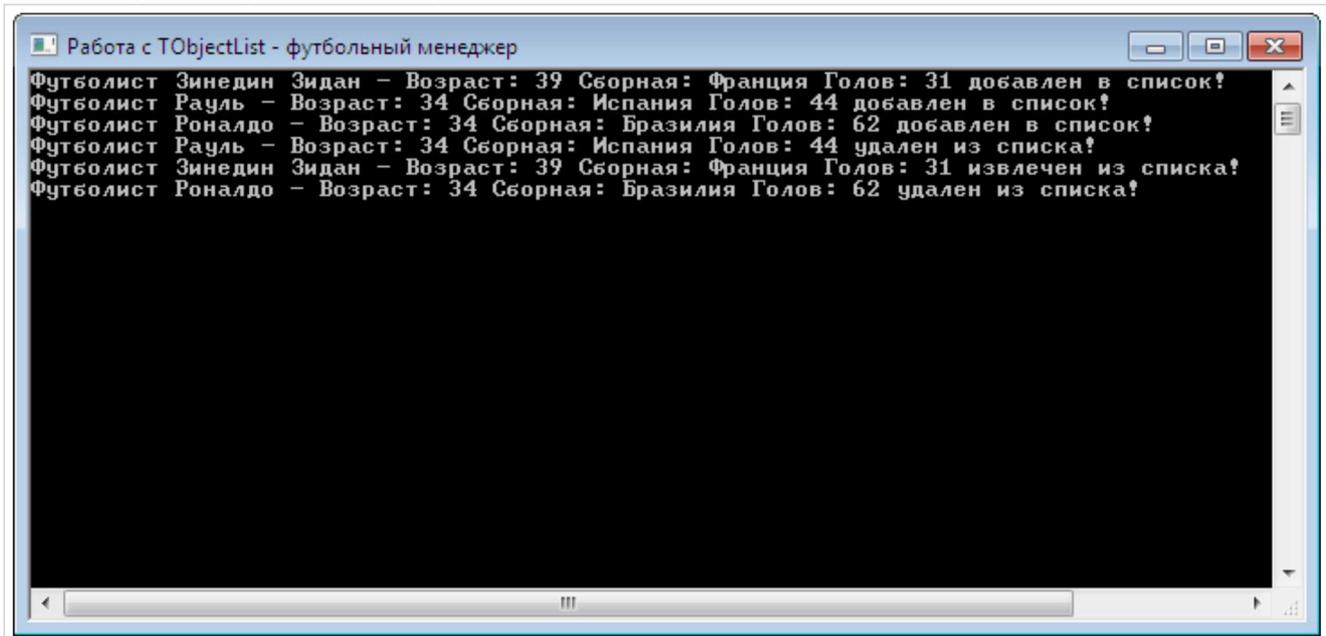


Рисунок 8 - Событие `OnNotify` в `TObjectList<T>`

3.4. `TStack<T>` и `TObjectStack<T>`

Стек представляет из себя обобщенную коллекцию элементов типа "последним пришел — первым вышел" (LIFO). Рассмотрим работу с ним на следующем примере:

Листинг 21 - Работа с `TStack<T>`

```

01. ...
02.
03. uses
04.     SysUtils, Generics.Collections;
05.
06. type
07.     // Будем печь блины, класть их на тарелку и брать последний =)
08.
09.     TPancakeType = (ptMeat, ptCherry, ptCurds);
10.
11.
12.     TPancake = record

```

```

13.     strict private
14.         const
15.             PANCAKE_TYPE_NAMES: array [TPancakeType] of string =
16.                 ('с мясом', 'с вишней', 'с творогом');
17.     public
18.         var
19.             PancakeType: TPancakeType;
20.         class function Create(PancakeType: TPancakeType): TPancake; static;
21.         function ToString: string;
22.     end;
23.
24. class function TPancake.Create(PancakeType: TPancakeType): TPancake;
25. begin
26.     Result.PancakeType := PancakeType;
27. end;
28.
29. function TPancake.ToString: string;
30. begin
31.     Result := Format('Блин %s', [PANCAKE_TYPE_NAMES[PancakeType]])
32. end;
33.
34. var
35.     PancakesPlate: TStack<TPancake>;
36.     Pancake: TPancake;
37.
38. begin
39.
40. ...
41.
42.     // "Создаем" тарелку с блинами
43.     PancakesPlate := TStack<TPancake>.Create;
44.
45.     // Испечем несколько блинов
46.     // Push - помещает элементы в стек
47.     PancakesPlate.Push(TPancake.Create(ptMeat));
48.     PancakesPlate.Push(TPancake.Create(ptCherry));
49.     PancakesPlate.Push(TPancake.Create(ptCherry));
50.     PancakesPlate.Push(TPancake.Create(ptCurds));
51.     PancakesPlate.Push(TPancake.Create(ptMeat));
52.
53.     // Съедем несколько блинов
54.     // Pop - извлекает элемент из стека
55.     Pancake := PancakesPlate.Pop;
56.     Writeln(Format('Съели блин (Pop): %s', [Pancake.ToString]));
57.     // Extract - аналогична Pop, но вызывает в OnNotify
58.     // с Action = cnExtracted вместо cnRemoved
59.     Pancake := PancakesPlate.Extract;
60.     Writeln(Format('Съели блин (Extract): %s', [Pancake.ToString]));
61.
62.     // Какой блин лежит последним?
63.     // Peek - возвращает последний элемент, но не извлекает его из стека
64.     Writeln(Format('Последний блин: %s', [PancakesPlate.Peek.ToString]));
65.
66.     // Покажем оставшиеся блины
67.     Writeln;
68.     Writeln(Format('Всего блинов: %d', [PancakesPlate.Count]));
69.     for Pancake in PancakesPlate do
70.         Writeln(Pancake.ToString);
71.
72.     // Доедаем все
73.     // Clear - очищает стек
74.     PancakesPlate.Clear;
75.
76.     FreeAndNil(PancakesPlate);
77.
78. ...

```

Результат работы приведен на *Рисунке 9*:

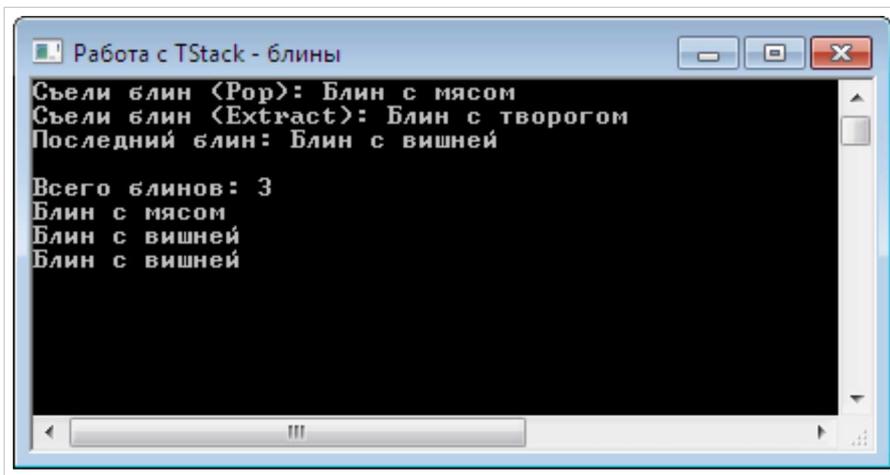


Рисунок 9 - Работа с TStack<T>

Событие OnNotify и функция ToArray работают также, как и в TList<T>.

3.5. TQueue<T> и TObjectQueue<T>

Очередь представляет из себя обобщенную коллекцию элементов, которая обслуживается по принципу "первым поступил — первым обслужен" (FIFO). Рассмотрим работу с очередью на следующем примере:

Листинг 22 - Работа с TQueue<T>

```

01. ...
02.
03. uses
04.   SysUtils, Generics.Collections;
05.
06. type
07.   // Будем обслуживать клиентов, стоящих за новой Delphi XE2 =)
08.
09.   TDelphiVersion = (dvStarter, dvProfessional, dvEnterprise, dvArchitect);
10.
11.   TCustomer = record
12.     strict private
13.       const
14.         DV_NAMES: array [TDelphiVersion] of string =
15.           ('Starter', 'Professional', 'Enterprise', 'Architect');
16.     public
17.       var
18.         DelphiVersion: TDelphiVersion;
19.       class function Create(DelphiVersion: TDelphiVersion): TCustomer; static;
20.       function ToString: string;
21.     end;
22.
23.   class function TCustomer.Create(DelphiVersion: TDelphiVersion): TCustomer;
24.   begin
25.     Result.DelphiVersion := DelphiVersion;
26.   end;
27.
28.   function TCustomer.ToString: string;
29.   begin
30.     Result := Format('Delphi XE2 %s', [DV_NAMES[DelphiVersion]])
31.   end;
32.
33.   var
34.     CustomerQueue: TQueue<TCustomer>;
35.     Customer: TCustomer;
36.
37.   begin
38.
39.   ...
40.
41.   // "Создаем" очередь в пункте продажи
42.   CustomerQueue := TQueue<TCustomer>.Create;
43.
44.

```

```

45. // Добавим несколько человек в очередь
46. // Enqueue - помещает элемент в очередь
47. CustomerQueue.Enqueue(TCustomer.Create(dvStarter));
48. CustomerQueue.Enqueue(TCustomer.Create(dvProfessional));
49. CustomerQueue.Enqueue(TCustomer.Create(dvProfessional));
50. CustomerQueue.Enqueue(TCustomer.Create(dvProfessional));
51. CustomerQueue.Enqueue(TCustomer.Create(dvEnterprise));
52. CustomerQueue.Enqueue(TCustomer.Create(dvEnterprise));
53. CustomerQueue.Enqueue(TCustomer.Create(dvArchitect));
54. CustomerQueue.Enqueue(TCustomer.Create(dvArchitect));
55.
56. // Обслужим часть покупателей
57. // Dequeue - извлекает элемент из очереди
58. Customer := CustomerQueue.Dequeue;
59. Writeln(Format('Продали (Dequeue): %s', [Customer.ToString]));
60. // Extract - аналогична Dequeue, но вызывает в OnNotify
61. // с Action = cnExtracted вместо cnRemoved
62. Customer := CustomerQueue.Extract;
63. Writeln(Format('Продали (Extract): %s', [Customer.ToString]));
64.
65. // За чем пришел следующий покупатель?
66. // Peek - возвращает первый элемент, но не извлекает его из очереди
67. Writeln(Format('Обслуживаемый покупатель пришел за %s',
68.                 [CustomerQueue.Peek.ToString]));
69.
70. // Оставшиеся покупатели
71. Writeln;
72. Writeln(Format('Всего покупателей осталось: %d', [CustomerQueue.Count]));
73. for Customer in CustomerQueue do
74.     Writeln(Customer.ToString);
75.
76. // Обслуживаем всех
77. // Clear - очищает очередь
78. CustomerQueue.Clear;
79.
80. FreeAndNil(CustomerQueue);
81.
82. ...

```

Результат работы приведен на *Рисунке 10*:

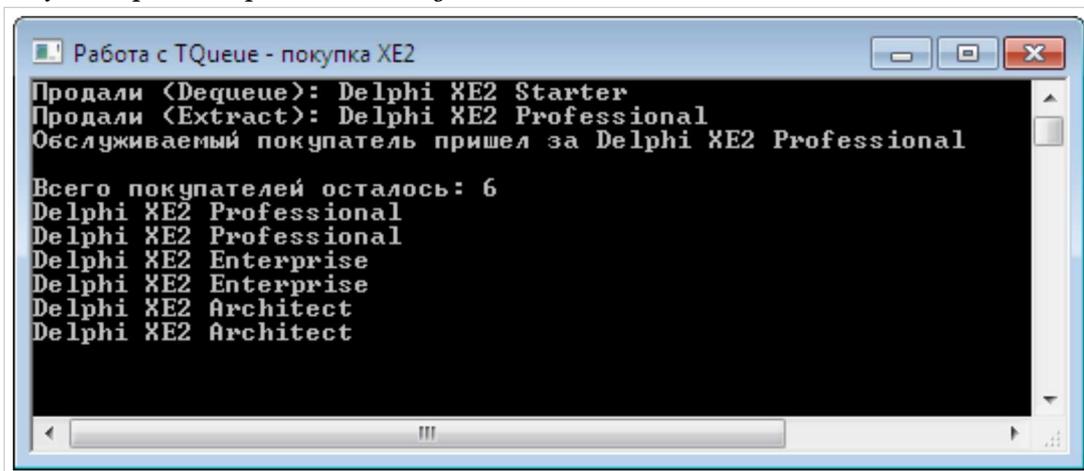


Рисунок 10 - Работа с TQueue<T>

Событие `OnNotify` и функция `ToArray` работают также, как и в `TList<T>`.

4. Заключение

Мы рассмотрели 2 основных модуля, предоставляющих классы для работы с дженериками, а также рассмотрели принципы и примеры работы с ними. Исходники проектов Вы сможете скачать в следующей Части.

Часть 3 - Приложение

1. Заключение

В материале были *кратко* рассмотрены дженерики и их основные возможности в Delphi, стандартные классы языка и примеры работы с ними. Для дальнейшего изучения настоятельно рекомендуется пройтись по ссылкам и изучить материалы, указанные ниже. Во-первых, это позволит углубить уже полученные знания, а во-вторых, открыть для себя новые возможности, пропущенные в настоящем материале сознательно для облегчения "стартового" понимания.

Общий совет один - старайтесь максимально использовать дженерики, используя их преимущества. Удачи!

Любые замечания, пожелания и благодарности по поводу статьи принимаются по почте (в заголовке указывайте тему статьи): keeper89@inbox.ru.

Адрес блога: <http://keeper89.blogspot.com>.

Для просмотра ссылок, приведенных в статье, и скачивания материалов, перейдите, пожалуйста, на ее интернет-версию: <http://keeper89.blogspot.com/2011/07/delphi.html>.

2. Исходники

В данном разделе приведены ссылки на исходные коды примеров, которые приводились в статье (с некоторыми изменениями). Вы можете скачать их и использовать по своему усмотрению.

Примечание: если у читателей будут интересные примеры использования дженериков, я буду рад их разместить в этом разделе, сохраняя ссылку на автора.

Класс(ы)	Описание	Ссылка
-	Реализация обобщенного класса	Скачать
-	Реализация обобщенного метода и записи	Скачать
TArray	Сортировка и поиск в одномерном целочисленном массиве	Скачать
	Сортировка двумерного массива	Скачать
TDictionary<T> и TObjectDictionary<T>	Работа со словарем	Скачать
TList<T> и TObjectList<T>	Работа со списком	Скачать
TStack<T> и TObjectStack<T>	Работа со стеком	Скачать
TQueue<T> и TObjectQueue<T>	Работа со очередью	Скачать
Скачать все примеры целиком		

3. Ссылки

1. Статья в PDF

- PDF-версия "[Используем дженерики в Delphi!](#)"

2. Дженерики

- Справочная система Delphi: обзор дженериков ([англ.](#)), Generics.Defaults ([англ.](#)), Generics.Collections ([англ.](#))
- М. Кэнтю Delphi Handbook 2009, раздел о дженериках ([англ.](#))
- Обобщённое программирование (generics) в Delphi 2009 для Win32 ([рус.](#), [англ.](#))
- Сравнивая дженерики в C#, C++ и Delphi(Win32) ([рус.](#), [англ.](#))

3. Другое

- Преимущества перехода на Delphi XE ([рус.](#), [англ.](#))