

## ГЛАВА 7

# Списки и коллекции

Практически любое приложение должно уметь выполнять ряд стандартных операций по обработке каких-либо данных. К ним относятся загрузка данных при открытии приложения, представление данных в удобном виде для использования внутри приложения, сохранение данных при завершении работы. Перечисленные действия необходимы и приложениям баз данных, и играм, и научным программам.

В принципе хранение и использование наборов значений можно обеспечить при помощи хорошо всем известных массивов. Однако их прямое использование требует от разработчика дополнительных усилий. Ведь при реализации программной логики необходимо добавлять в массив новые элементы, изменять существующие и удалять ненужные. Кроме этого, часто бывает необходимо найти элемент массива по значению. Все эти операции стандартны и повторяются для наборов любых типов данных.

Для решения перечисленных задач в Delphi доступны для использования специальные классы. Помимо хранения наборов значений в них реализованы свойства, позволяющие контролировать состояние списка и методы, обеспечивающие редактирование списка и поиск в нем отдельных элементов.

Для загрузки и сохранения данных используются потоки — классы, инкапсулирующие механизмы доступа к различным хранилищам информации — файлам, памяти и т. д. Их общим предком является класс `TStream`.

Для работы со строковыми списками предназначены классы `TStrings` и `TStringList`.

Любые типы данных можно заносить в список указателей, который реализован в классе `TList`.

Использование наборов объектов (широко применяются в классах `VCL`), которые называются *коллекциями*, осуществляется при помощи классов `TCollection` и `TCollectionItem`.

В этой главе рассматриваются следующие вопросы:

- ☐ что такое список; как устроено основное свойство всех списков, объединяющее его элементы;
- ☐ добавление, изменение и удаление элементов списка;
- ☐ поиск заданного элемента;
- ☐ механизм выделения памяти под элементы списка;
- ☐ список строк;
- ☐ список указателей;
- ☐ чем отличается коллекция от списка;
- ☐ коллекции;
- ☐ использование потоков.

## Список строк

Строковый тип данных широко используется программистами. Во-первых, многие данные действительно необходимо представлять при помощи этого типа. Во-вторых, множество функций преобразования типов позволяют представлять числовые типы в виде строк, избегая тем самым проблем с несовместимостью типов.

По этой причине в первую очередь мы займемся изучением списка строк, который инкапсулирован в классах `TStrings` и `TStringList`. Первый класс является абстрактным и служит платформой для создания реально работающих потомков. Второй класс реализует вполне работоспособный список строк. Рассмотрим эти классы подробнее.

## Класс *TStrings*

Класс `TStrings` является базовым классом, который обеспечивает потомков основными свойствами и методами, позволяющими создавать работоспособные списки строк. Его прямым предком является класс `TPersistent`.

Класс `TStrings` реализует все вспомогательные свойства и методы, которые обеспечивают управление списком. При этом методы, непосредственно добавляющие и удаляющие элементы списка, не реализованы и объявлены как абстрактные.

### Внимание!

Попытка прямого использования в приложении экземпляра класса `TStrings` вызовет ошибку применения абстрактного класса на этапе выполнения программы, а именно при попытке заполнить список значениями. Простая замена

типа объектной переменной списка на `TStringList` делает приложение полностью работоспособным без какого-либо дополнительного изменения исходного кода.

Классы-наследники должны перекрывать методы добавления и удаления элементов списка. Реализованный в Delphi класс `TStringList` практически полностью повторяет функциональность предка, добавляя лишь несколько новых свойств и методов. Поэтому мы не станем останавливаться подробнее на классе `TStrings`, а перейдем сразу к его работоспособному потомку `TStringList`.

## Класс *TStringList*

Класс `TStringList` обеспечивает реальное использование списков строк в приложении. По существу, класс представляет собой оболочку вокруг динамического массива значений списка, представленного свойством `strings`. Объявление свойства (унаследованное от `TStrings`) выглядит так:

```
property Strings[Index: Integer]: string read Get write Put; default;
```

Для работы со свойством используются внутренние методы `Get` и `Put`, в которых применяется внутренняя переменная `FList`:

```
type
  PStringItem = ^TStringItem;
  TStringItem = record
    FString: string;
    FObject: TObject;
  end;
  PStringItemList = ^TStringItemList;
  TStringItemList = array[0..MaxListSize] of TStringItem;

FList: PStringItemList;
```

Из ее объявления видно, что список строк представляет собой динамический массив записей `TStringItem`. Эта запись позволяет объединить саму строку и связанный с ней объект.

Максимальный размер списка ограничен константой

```
MaxListSize = Maxint div 16;
```

значение которой после нехитрых вычислений составит 134 217 727. Таким образом, видно, что строковый список Delphi теоретически конечен, хотя на практике гораздо чаще размер списка ограничивается размером доступной памяти.

Обращение к отдельному элементу списка может осуществляться через свойство `strings` таким образом:

```
SomeStrings.Strings[i] := Edit1.Text;
```

или так:

```
SomeStrings[i] := Edit1.Text;
```

Оба способа равноценны.

При помощи простого присваивания можно задавать новые значения только тогда, когда элемент уже создан. Для добавления нового элемента используются методы `Add` и `AddStrings`.

**Функция**

```
function Add(const S: string): Integer;
```

добавляет в конец списка новый элемент, присваивая ему значение `s` и возвращая индекс нового элемента в списке.

**Метод**

```
procedure Append(const S: string);
```

просто вызывает функцию `Add`. Единственное отличие заключается в том, что метод не возвращает индекс нового элемента.

**Метод**

```
procedure AddStrings(Strings: TStrings);
```

добавляет к списку целый набор новых элементов, которые должны быть заданы другим списком, передаваемым в параметре `strings`.

При необходимости можно добавить новый элемент в произвольное место списка. Для этого применяется метод

```
procedure Insert(Index: Integer; const S: string);
```

который вставляет элемент `s` на место элемента с индексом `index`. При этом все указанные элементы смещаются на одну позицию вниз.

Для удаления элемента списка используется метод

```
procedure Delete(Index: Integer);
```

**Метод**

```
procedure Move(CurIndex, NewIndex: Integer);
```

перемещает элемент, заданный индексом `CurIndex`, на новую позицию, заданную индексом `NewIndex`.

**А метод**

```
procedure Exchange(Index1, Index2: Integer);
```

меняет местами элементы с индексами `Index1` и `Index2`.

Довольно часто в списках размещается строковая информация следующего вида:

```
'Name=Value'
```

В качестве примера можно привести строки из файлов INI или системного реестра. Специально для таких случаев в списке предусмотрено представление строк в двух свойствах. В свойстве `Names` содержится текст до знака равенства. В свойстве `values` содержится текст после знака равенства по умолчанию. Однако символ-разделитель можно заменить на любой другой, используя свойство

```
property NameValueSeparator: Char;
```

Доступ к значениям свойства `Values` осуществляется по значению. Например, если в списке есть строка

```
City=Saint-Petersburg
```

то значение свойства `value` будет равно

```
Value['City'] = 'Saint-Petersburg'
```

Кроме этого, значение свойства `value` можно получить, если известен его индекс:

```
property ValueFormIndex[Index: Integer]: string;
```

Как видно из объявления внутреннего списка `FList` (см. выше), с каждым элементом списка можно связать любой объект. Для этого используется свойство

```
property Objects[Index: Integer]: TObject;
```

Свойство `strings` элемента и свойство `Objects` связанного с ним объекта имеют одинаковые индексы. Если строка не имеет связанного объекта, то свойство `objects` равно `Nil`. Один объект может быть связан с несколькими строками списка одновременно.

Чаше всего объекты нужны для того, чтобы хранить для каждого элемента дополнительную информацию. Например, в списке городов для каждого элемента можно дополнительно хранить население, площадь, административный статус и т. д. Для этого можно создать примерно такой класс:

```
TCityProps = class(TObject)
    Square: LongInt;
    Population: LongInt;
    Status: String;
end;
```

Для того чтобы добавить к строке из списка объект, используется метод `AddObject`:

```
function AddObject(const S: string; AObject: TObject): Integer; virtual;
```

Обратите внимание, что в параметре AObject необходимо передавать указатель на объект. Проще всего это сделать таким образом:

```
SomeStrings.AddObject('SomeItem', TCityProps.Create);
```

Или же так:

```
var SPb: TCityProps;

SPb := TCityProps.Create; {Создание объекта}
SPb.Population := 5000000;

SomeStrings.Strings[i] := 'Санкт-Петербург';
SomeStrings.Objects[i] := SPb;      {Связывание объекта и строки}
```

Можно поступить и подобным образом (помните, что строка уже должна существовать):

```
SomeStrings.Strings[i] := 'Санкт-Петербург';
SomeStrings.Objects[i] := TCityProps.Create;
(SomeStrings.Objects[i] as TCityProps).Population := 5000000;
```

Аналогично методу insert, элемент и связанный с ним объект можно вставить в произвольное место списка методом

```
procedure InsertObject(Index: Integer; const S: string; AObject: TObject);
```

При перемещении методом Move вместе с элементом переносится и указатель на связанный объект.

Обратите внимание на две особенности, связанные с удалением указателей на объекты и самих связанных объектов.

При удалении элемента списка удаляется только указатель на объект, а сам объект остается в памяти. Для его уничтожения следует предпринять дополнительные усилия:

```
for i := 0 to SomeList.Count - 1 do
  SomeList.Objects[i].Destroy;
```

Если при удалении связанного объекта необходимо выполнить некоторые действия, предусмотренные в деструкторе, приведение типов

```
TCityProps(SomeList.Objects[i]).Destroy;
```

выполнять не обязательно — нужный деструктор будет вызван автоматически, хотя в данном случае приведение типов ошибкой не является.

### Метод

```
procedure Clear; override;
```

полностью очищает список, удаляя все его элементы.

Помимо перечисленных, класс `TStringList` обладает рядом дополнительных свойств и методов. Вспомогательные свойства класса обеспечивают разработчика информацией о состоянии списка. Дополнительные методы осуществляют поиск в списке и взаимодействие с файлами и потоками.

Свойство только для чтения

```
property Count: Integer;
```

возвращает число элементов списка.

Так как основу списка составляет динамический массив, то для него в процессе работы должна выделяться память. При добавлении в список новой строки память для нее выделяется автоматически. Свойство

```
property Capacity: Integer;
```

определяет число строк, для которых выделена память. Вы можете самостоятельно управлять этим параметром, помня при этом, что значение `capacity` всегда должно быть больше или равно значению `Count`.

Свойство

```
property Duplicates: TDuplicates;
```

определяет, можно ли добавлять в список повторные значения.

Тип

```
type
```

```
TDuplicates = (dupIgnore, dupAccept, dupError);
```

определяет реакцию списка на добавление повторного элемента:

- ☐ `dupIgnore` — запрещает добавление повторных элементов;
- ☒ `dupAccept` — разрешает добавление повторных элементов;
- ☐ `dupError` — запрещает добавление повторных элементов и генерирует исключительную ситуацию.

Класс `TStringList` немислимо представить себе без возможностей сортировки. Если вас удовлетворит обычная сортировка, то для этого можно использовать свойство `sorted` (сортировка выполняется при значении `True`) или метод `sort`. Под "обычной" имеется в виду сортировка по тексту строк с использованием функции `AnsiCompareStr` (т. е. с учетом национальных символов, в порядке возрастания). Если вы хотите отсортировать список по другому критерию, к вашим услугам метод:

type

```
TStringListSortCompare = function(List: TStringList; Index1,
Index2: Integer): Integer;
procedure CustomSort(Compare: TStringListSortCompare);
```

Чтобы отсортировать список, вы должны описать функцию сравнения двух элементов с индексами Index1 и Index2, которая должна возвращать следующие результаты:

- 1 — если элемент с индексом Index1 вы хотите поместить впереди элемента Index2;
- 0 — если они равны;
- -1 — если элемент с индексом index1 вы хотите поместить после элемента Index2.

Для описанного выше примера с объектом-городом нужны три процедуры:

```
function SortByStatus(List: TStringList; Index1, Index2: Integer):
Integer;
begin
Result := AnsiCompareStr((List.Objects[Index1] as TCityProps).Status,
(List.Objects[Index2] as TCityProps).Status);
end;

function SortBySquare(List: TStringList; Index1, Index2: Integer):
Integer;
begin
if (List.Objects[Index1] as TCityProps).Square <
(List.Objects[Index2] as TCityProps).Square) then Result := -1
else if (List.Objects[Index1] as TCityProps).Square =
(List.Objects[Index2] as TCityProps).Square then Result := 0
else Result := 1;
end;

function SortByPopulation(List: TStringList; Index1, Index2: Integer):
Integer;
begin
if (List.Objects[Index1] as TCityProps).Population <
(List.Objects[Index2] as TCityProps).Population then Result := -1
else
if (List.Objects[Index1] as TCityProps).Population =
(List.Objects[Index2] as TCityProps).Population
then Result := 0
else Result := 1;
end;
```

Передаем одну из процедур в метод CustomSort:

```
Cities.CustomSort(SortByPopulation);
```



Для поиска нужного элемента используется метод

```
function Find(const S: string; var Index: Integer): Boolean;
```

В параметре *s* передается значение для поиска. В случае успеха функция возвращает значение *True*, а в параметре *index* содержится индекс найденного элемента.

Метод

```
function IndexOf(const S: string): Integer;
```

возвращает индекс найденного элемента *s*. Иначе функция возвращает  $-1$ .

Метод

```
function IndexOfName(const Name: string): Integer;
```

возвращает индекс найденного элемента, для которого свойство *Names* совпадает со значением параметра *Name*.

Для поиска связанных объектов используется метод

```
function IndexOfObject(AObject: TObject): Integer;
```

В качестве параметра *AObject* должна передаваться ссылка на искомый объект.

А свойство

```
property CaseSensitive: Boolean;
```

включает или отключает режим поиска и сортировки с учетом регистра символов.

Помимо свойства *strings*, содержимое списка можно получить при помощи свойств

```
property Text: string;
```

и

```
property CommaText: string;
```

Они представляют все строки списка в виде одной строки. При этом в первом свойстве элементы списка разделены символами возврата каретки и переноса строки. Во втором свойстве строки заключены в двойные кавычки и разделены запятыми или пробелами. Так, для списка городов (Москва, Петербург, Одесса) свойство *Text* будет равно

```
Москва#D#$APетербург#D#$AOдесса
```

а СВОЙСТВО *CommaText* равно

```
"Москва", "Петербург", "Одесса".
```

Важно иметь в виду, что эти свойства доступны не только по чтению, но и по записи. Так что заполнить список вы сможете не только циклически,

вызывая и используя методы `Add` или `insert`, но и одним-единственным Присвоением Значения свойствам `Text` ИЛИ `CommaText`.

Список может взаимодействовать с другими экземплярами класса `TStringList`.

Широко распространенный метод

```
procedure Assign(Source: TPersistent);
```

полностью переносит список `Source` в данный.

Метод

```
function Equals(Strings: TStrings): Boolean;
```

возвращает значение `True`, если элементы списка `strings` полностью совпадают с элементами данного списка.

Список можно загрузить из файла или потока. Для этого используются методы

```
procedure LoadFromFile(const FileName: string);
```

и

```
procedure LoadFromStream(Stream: TStream);
```

Сохранение списка выполняется методами

```
procedure SaveToFile(const FileName: string);
```

и

```
procedure SaveToStream(Stream: TStream);
```

Перед изменением списка вы можете получить управление, описав обработчик события

```
property OnChange: TNotifyEvent;
```

а после изменения

```
property OnChanging: TNotifyEvent;
```

На дискете, прилагаемой к этой книге, вы можете ознакомиться с примером использования СПИСКОВ СТРОК `DemoStrings`.

## Список указателей

Для хранения списка указателей на размещенные в адресном пространстве структуры (объекты, динамические массивы, переменные) предназначен класс `TList`. Так же, как и список строк `TStringList`, список указателей обеспечивает эффективную работу с элементами списка.

## Класс *TList*

Основой класса *TList* является список указателей. Сам список представляет собой динамический массив указателей, к которому можно обратиться через индексированное свойство

```
property Items[Index: Integer]: Pointer;
```

Нумерация элементов начинается с нуля.

Прямой доступ к элементам массива возможен через свойство

```
type
```

```
  PPointerList = ^TPointerList;
```

```
  TPointerList = array[0..MaxListSize-1] of Pointer;
```

```
property List: PPointerList;
```

которое имеет атрибут "только для чтения".

Так как элементы списка являются указателями на некоторые структуры, прямое обращение к составным частям этих структур через свойство *items* невозможно. Как это можно сделать, рассказывается ниже в примере.

### **Примечание**

В списке могут содержаться указатели на разнородные структуры. Не обязательно хранить в списке только указатели на объекты или указатели на записи.

Реализованные в классе *TList* операции со списком обеспечивают потребности разработчика и совпадают с операциями списка строк.

Для добавления в конец списка нового указателя используется метод

```
function Add(Item: Pointer): Integer;
```

Прямое присваивание значения элементу, который еще не создан при помощи метода *Add*, вызовет ошибку времени выполнения.

Новый указатель можно добавить в нужное место списка. Для этого используется метод

```
procedure Insert(Index: Integer; Item: Pointer);
```

В параметре *index* указывается необходимый порядковый номер в списке.

Перенос существующего элемента на новое место осуществляется методом

```
procedure Move(CurIndex, NewIndex: Integer);
```

Параметр *CurIndex* определяет старое положение указателя. Параметр *NewIndex* задает новое его положение.

Также можно поменять местами два элемента, определяемые параметрами *Index1* и *Index2*:

```
procedure Exchange(Index1, Index2: Integer);
```

Для удаления указателей из списка используются два метода. Если известен индекс, применяется метод

```
procedure Delete(Index: Integer);
```

Если известен сам указатель, используется метод

```
function Remove(Item: Pointer): Integer;
```

Эти методы не уменьшают объем памяти, выделенной под список. При необходимости сделать это следует использовать свойство `capacity`. Также существует метод `Expand`, который увеличивает отведенную память автоматически в зависимости от текущего размера списка.

```
function Expand: TList;
```

Для того чтобы метод сработал, необходимо, чтобы `Count = Capacity`. Алгоритм работы метода представлен в табл. 7.1.

**Таблица 7.1.** Алгоритм увеличения памяти списка

Значение свойства <code>Capacity</code>	На сколько увеличится свойство <code>Capacity</code>
<4	4
4 8	8
>8	16

#### Метод

```
procedure Clear; dynamic;
```

используется для удаления всех элементов списка сразу.

Для поиска указателя по его значению используется метод

```
function IndexOf(Item: Pointer): Integer;
```

Метод возвращает индекс найденного элемента в списке. При неудачном поиске возвращается `-1`.

Для сортировки элементов списка применяется метод

```
type TListSortCompare = function (Item1, Item2: Pointer): Integer;
procedure Sort(Compare: TListSortCompare);
```

Так как состав структуры, на которую указывает элемент списка, невозможно заранее обобщить, разработка процедуры, осуществляющей сортировку, возлагается на программиста. Метод `Sort` лишь обеспечивает попарное сравнение указателей на основе созданного программистом алгоритма (пример сортировки см. выше в разд. "Класс `TStringList`").

Полностью все свойства и методы класса `TList` представлены в табл. 7.2.

**Таблица 7.2.** Свойства и методы класса `TList`

Объявление	Описание
<code>property Capacity: Integer;</code>	Определяет число строк, для которых выделена память
<code>property Count: Integer;</code>	Возвращает число строк в списке
<code>property Items[Index: Integer]: Pointer;</code>	Список указателей
<code>type TPointerList = array[0..MaxListSize-1] of Pointer; PPointerList = ^TPointerList;</code>	Динамический массив указателей
<code>property List: PPointerList;</code>	
<code>function Add(Item: Pointer): Integer;</code>	Добавляет к списку новый указатель
<code>procedure Clear; dynamic;</code>	Полностью очищает список
<code>procedure Delete(Index: Integer);</code>	Удаляет указатель с индексом <code>Index</code>
<code>class procedure Error(const Msg: string; Data: Integer); virtual;</code>	Генерирует исключительную ситуацию <code>EListError</code> . Сообщение об ошибке создается из форматирующей строки <code>Msg</code> и числового параметра <code>Data</code>
<code>procedure Exchange(Index1, Index2: Integer);</code>	Меняет местами указатели с индексами <code>Index1</code> и <code>Index2</code>
<code>function Expand: TList;</code>	Увеличивает размер памяти, отведенной под список
<code>function First: Pointer;</code>	Возвращает первый указатель из списка
<code>function IndexOf(Item: Pointer): Integer;</code>	Возвращает индекс указателя, заданного параметром <code>Item</code>
<code>procedure Insert(Index: Integer; Item: Pointer);</code>	Вставляет новый элемент <code>Item</code> в позицию <code>Index</code>
<code>function Last: Pointer;</code>	Возвращает последний указатель в списке
<code>procedure Move(CurIndex, NewIndex: Integer);</code>	Перемещает элемент списка на новое место

Таблица 7.2 (окончание)

Объявление	Описание
<code>procedure Pack;</code>	Удаляет из списка все пустые (Nil) указатели
<code>function Remove(Item: Pointer): Integer;</code>	Удаляет из списка указатель Item
<code>type TListSortCompare = function (Item1, Item2: Pointer): Integer;</code>	Сортирует элементы списка
<code>procedure Sort(Compare: TListSortCompare);</code>	

## Пример использования списка указателей

Рассмотрим использование списков указателей на примере приложения DemoList. При щелчке мышью на форме приложения отображается точка, которой присваивается порядковый номер. Одновременно координаты и номер точки записываются в соответствующие свойства создаваемого экземпляра класса `TMyPixel`. Указатель на этот объект передается в новый элемент списка `pixList`.

В результате после очистки формы всю последовательность точек можно восстановить, используя указатели на объекты точек из списка.

Список точек можно отсортировать по координате X в порядке возрастания.

⚡ Листинг 7.1. Модуль главной формы проекта DemoList

```
unit Main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, Buttons;

type
  TMainForm = class(TForm)
    ListBtn: TBitBtn;
    ClearBtn: TBitBtn;
    DelBtn: TBitBtn;
    SortBtn: TBitBtn;
    procedure FormCreate(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
```

```

    procedure FormMouseDown(Sender: TObject; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
    procedure ListBtnClick(Sender: TObject);
    procedure ClearBtnClick(Sender: TObject);
    procedure DelBtnClick(Sender: TObject);
    procedure SortBtnClick(Sender: TObject);
private
    PixList: TList;
    PixNum: Integer;
public
    { Public declarations }
end;

TMyPixel = class(TObject)
    FX:      Integer;
    FY:      Integer;
    FText:   Integer;
    constructor Create(X, Y, Num: Integer);
    procedure SetPixel;
end;

var
    MainForm: TMainForm;

implementation

{$R *.DFM}

const PixColor = clRed;

var CurPixel: TMyPixel;

constructor TMyPixel.Create(X, Y, Num: Integer);
begin
    inherited Create;
    FX := X;
    FY := Y;
    FText := Num;
    SetPixel;
end;

procedure TMyPixel.SetPixel;
begin
    MainForm.Canvas.PolyLine([Point(FX, FY), Point(FX, FY)]);
    MainForm.Canvas.TextOut(FX + 1, FY + 1, IntToStr(FText));
end;

```

```

function PixCompare(Item1, Item2: Pointer): Integer;
var Pix1, Pix2: TMyPixel;
begin
    Pix1 := Item1;
    Pix2 := Item2;
    Result := Pix1.FX - Pix2.FX;
end;

procedure TMainForm.FormCreate(Sender: TObject);
begin
    PixList := TList.Create;
    PixNum := 1;           {Счетчик точек}
    Canvas.Pen.Color := PixColor; {Цвет точки}
    Canvas.Pen.Width := 3;   {Размер точки}
    Canvas.Brush.Color := Color; {Цвет фона текста равен цвету формы}
end;

procedure TMainForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    PixList.Free;
end;

procedure TMainForm.FormMouseDown(Sender: TObject; Button: TMouseButton;
    Shift: TShiftState; X, Y: Integer);
begin
    PixList.Add(TMyPixel.Create(X, Y, PixNum));
    Inc(PixNum);
end;

procedure TMainForm.ListBtnClick(Sender: TObject);
var i: Integer;
begin
    with PixList do
        for i := 0 to Count - 1 do
            begin
                CurPixel := Items[i];
                CurPixel.SetPixel;
            end;
        end;
end;

procedure TMainForm.ClearBtnClick(Sender: TObject);
begin
    Canvas.FillRect(Rect(0, 0, Width, Height));
end;

```



```
procedure TMainForm.DelBtnClick(Sender: TObject);
begin
  PixList.Clear;
  PixNum := 1;
end;

procedure TMainForm.SortBtnClick(Sender: TObject);
var i: Integer;
begin
  PixList.Sort(PixCompare);
  with PixList do
    for i := 0 to Count - 1 do TMyPixel(Items[i]).FText := i + 1;
  end;
end.
```

Класс TMyPixel обеспечивает хранение координат точки и ее порядковый номер в серии. Эти параметры передаются в конструктор класса. Метод setPixel обеспечивает отрисовку точки на канве формы (см. гл. 10).

Экземпляр класса создается для каждой новой точки при щелчке кнопкой мыши в методе-обработчике FormMouseDown. Здесь же указатель на новый объект сохраняется в создаваемом при помощи метода Add элементе списка PixList. Таким образом, программа "запоминает" расположение и порядок следования точек.

Метод-обработчик ListBtnClick обеспечивает отображение точек. Для этого в цикле текущий указатель списка передается в переменную объектного типа CurPixel, т. е. в этой переменной по очереди "побывают" все созданные объекты, указатели на которые хранятся в списке.

Это сделано для того, чтобы получить доступ к свойствам объектов (непосредственно через указатель этого сделать нельзя). Второй способ приведения типа рассмотрен в методе-обработчике SortBtnClick.

Перед вторичным отображением точек необходимо очистить поверхность формы. Эту операцию выполняет метод-обработчик ClearBtnClick.

Список точек можно отсортировать по координате X в порядке возрастания. Для этого в методе-обработчике SortBtnClick вызывается метод sort списка PixList. В параметре метода (переменная процедурного типа) передается функция PixCompare, которая обеспечивает инкапсулированный в методе sort механизм перебора элементов списка алгоритмом принятия решения о старшинстве двух соседних элементов.

Если функция возвращает положительное число, то элемент Item1 больше элемента Item2. Если результат отрицательный, то Item1 меньше, чем Item2. Если элементы равны, функция должна возвращать ноль.

В нашем случае сравнивались координаты X двух точек. В результате такой сортировки по возрастанию объекты оказались расположены так, что первый элемент списка указывает на объект с минимальной координатой X, ; последний — на объект с максимальной координатой X.

После сортировки осталось заново пронумеровать все точки. Это делает цикл в методе-обработчике `SortBtnClick`. Обратите внимание на примененный в этом случае способ приведения типа, обеспечивающий обращение к свойствам экземпляров класса `TMyPixel`.

Метод-обработчик `DelBtnClick` обеспечивает полную очистку списка `PixList`.

## Коллекции

*Коллекция* представляет собой разновидность списка указателей, оптимизированную для работы с объектами определенного вида. Сама коллекция инкапсулирована в классе `TCollection`. Элемент коллекции должен быть экземпляром класса, унаследованного от класса `TCollectionItem`. Это облегчает программирование и позволяет обращаться к свойствам и методам объектов напрямую.

Коллекции объектов широко используются в компонентах VCL. Например панели компонента `TCoolBar` (см. гл. 5) объединены в коллекцию. Класс `TCoolBands`, объединяющий панели, является наследником класса `TCollection`. А отдельная панель — экземпляром класса `TCoolBar`, происходящего от класса `TCollectionItem`.

Поэтому знание свойств и методов классов коллекции позволит успешно использовать их при работе со многими компонентами (`TDBGrid`, `TListView`, `TStatusBar`, `TCoolBar` и т. д.).

Для работы с коллекцией, независимо от инкапсулирующего ее компонента, применяется специализированный Редактор коллекции (рис. 7.1), набор

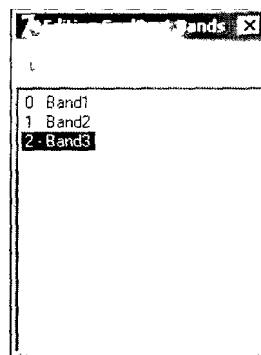


Рис. 7.1. Редактор коллекции

элементов управления которого может немного изменяться для разных компонентов.

Список Редактора объединяет элементы коллекции. При выборе одной строки из списка свойства объекта коллекции становятся доступны в Инспекторе объектов. В список можно добавлять новые элементы и удалять существующие, а также менять их взаимное положение.

Примеры использования коллекций представлены при описании соответствующих компонентов.

## Класс *TCollection*

Класс *TCollection* является оболочкой коллекции, обеспечивая разработчика набором свойств и методов для управления ею (табл. 7.3).

Сама коллекция содержится в свойстве

```
property Items[Index: Integer]: TCollectionItem;
```

Полное объявление свойства в классе выглядит следующим образом:

```
property Items[Index: Integer]: TCollectionItem read GetItem write SetItem;
```

Методы *GetItem* и *SetItem* обращаются к внутреннему полю *FItems*:

```
FItems: TList;
```

Именно оно хранит коллекцию объектов во время выполнения. Отсюда следует, что коллекция представляет собой список указателей на экземпляры класса *TCollectionItem* или его наследника. Класс *TCollection* обеспечивает удобство использования элементов списка.

**Таблица 7.3.** Свойства и методы класса *TCollection*

Объявление	Описание
<code>property Count: Integer;</code>	Возвращает число элементов коллекции
<code>type TCollectionItemClass = class of TCollectionItem;</code> <code>property ItemClass: TCollectionItemClass;</code>	Возвращает класс-наследник <i>TCollectionItem</i> , экземпляры которого собраны в коллекции
<code>property Items[Index: Integer]: TCollectionItem;</code>	Коллекция экземпляров класса
<code>function Add: TCollectionItem;</code>	Добавляет к коллекции новый экземпляр класса
<code>procedure Assign(Source: TPersistent); override;</code>	Копирует коллекцию из объекта <i>Source</i> в данный объект

Таблица 7.3 (окончание)

Объявление	Описание
<code>procedure BeginUpdate; virtual;</code>	Отменяет перерисовку коллекции. Используется при внесении изменений в коллекцию
<code>procedure Clear;</code>	Удаляет из коллекции все элементы
<code>procedure EndUpdate; virtual;</code>	Отменяет действие метода <code>BeginUpdate</code>
<code>function FindItemID(ID: Integer): TCollectionItem;</code>	Возвращает объект коллекции с номером ID
<code>function GetNamePath: string; override;</code>	Возвращает имя класса коллекции во время выполнения, если коллекция не имеет владельца. Иначе возвращает название свойства класса, владеющего коллекцией
<code>function Insert(Index: Integer): TCollectionItem;</code>	Вставляет в коллекцию новый объект на место с номером Index

## Класс *TCollectionItem*

Класс *TCollectionItem* инкапсулирует основные свойства и методы элемента коллекции (табл. 7.4). Свойства класса обеспечивают хранение информации о расположении элемента в коллекции.

Таблица 7.4. Свойства и методы класса *TCollectionItem*

Объявление	Описание
<code>property Collection: TCollection;</code>	Содержит экземпляр класса коллекции, которой принадлежит данный элемент
<code>property DisplayName: string;</code>	Содержит имя элемента, которое представляет его в Редакторе коллекции
<code>property ID: Integer;</code>	Содержит уникальный номер элемента в коллекции, который не может изменяться
<code>property Index: Integer;</code>	Содержит порядковый номер элемента в коллекции. Он соответствует положению элемента в списке и может изменяться

## Резюме

Списки, объединяющие элементы различных типов, играют важную роль при создании программной логики приложения. В Delphi используются три основных вида списков.

- Классы `TStrings` и `TStringList` обеспечивают применение списков строк.
- Класс `TList` инкапсулирует список указателей.
- Классы `TCollection` и `TCollectionItem` позволяют применять в компонентах и программном коде коллекции группы однородных объектов.

В среде Delphi вы можете найти еще много полезных классов общего применения. В модуле `CLASSES.PAS` есть класс `TBits`, обеспечивающий побитное чтение и запись информации. В модуле `CONTNRS.PAS` есть классы `TStack` и `TQueue` (стек и очередь), а также потомки `TList` — `TClassList`, `TComponentList` и т. д. Они помогут вам решать типовые задачи быстро и без "изобретения велосипеда".