Объектно-ориентированное программирование

По сравнению с традиционными способами программирования ООП обладает рядом преимуществ. Главное из них заключается в том, что эта концепция в наибольшей степени соответствует внутренней логике функционирования операционной системы (ОС) Windows. Программа, состоящая из отдельных объектов, отлично приспособлена к реагированию на события, происходящие в ОС. К другим преимуществам ООП можно отнести большую надежность кода и возможность повторного использования отработанных объектов.

В этой главе рассматриваются способы реализации основных механизмов ООП в Object Pascal и Delphi:

- понятия объекта, класса и компонента;
- основные механизмы ООП: инкапсуляция, наследование и полиморфизм;
- особенности реализации объектов;
- взаимодействие свойств и методов.

Объект и класс

Перед началом работы необходимо ввести основные понятия и определения.

Классом в Object Pascal называется структура языка, которая может иметь в своем составе переменные, функции и процедуры. Переменные в зависимости от предназначения именуются *полями* или *свойствами (см. ниже)*. Процедуры и функции класса – *методами.* Соответствующий классу тип будем называть *объектным типом:*

```
type
   TMyObject = class(TObject)
      MyField: Integer;
      function MyMethod: Integer;
   end;
```

В этом примере описан класс TMyObject, содержащий поле MyField и метод MyMethod.

Поля объекта аналогичны полям записи (record). Это данные, уникальные для каждого созданного в программе экземпляра класса. Описанный здесь класс TMyObject имеет одно поле – MyField.

Методы – это процедуры и функции, описанные внутри класса и предназначенные для операций над его полями. В состав класса входит указатель на специальную таблицу, где содержится вся информация, нужная для вызова методов. От обычных процедур и функций методы отличаются тем, что им при вызове передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут поля именно того объекта, который

вызвал метод. Внутри метода указатель на вызвавший его объект доступен под зарезервированным именем Self.

Понятие свойства будет подробно рассмотрено ниже. Пока можно определить его как поле, доступное для чтения и записи не напрямую, а через соответствующие методы.

Классы могут быть описаны либо в секции интерфейса модуля, либо на верхнем уровне вложенности секции реализации. Не допускается описание классов "где попало", т. е. внутри процедур и других блоков кода.

Разрешено опережающее объявление классов, как в следующем примере:

```
type
   TFirstObject = class;
   TSecondObject = class(TObject)
   First : TFirstObject;
      . . . . . .
   end;
   TFirstObject = class(TObject)
      . . . . . .
   end;
```

Чтобы использовать класс в программе, нужно, как минимум, объявить переменную этого типа. Переменная объектного типа называется *экземпляром класса* или *объектом:*

```
var
   AMyObject: TMyObject;
```

До введения термина "класс" в языке Pascal существовала двусмысленность определения "объект", который мог обозначать и тип, и переменную этого типа. Теперь же существует четкая граница: класс – это описание, объект – то, что создано в соответствии с этим описанием.

Как создаются и уничтожаются объекты?

Те, кто раньше использовал ООП в работе на C++ и особенно в Turbo Pascal, будьте внимательны: в Object Pascal экземпляры объектов могут быть только динамическими. Это означает, что в приведенном выше фрагменте переменная AMyObject на самом деле является указателем, содержащим адрес объекта.

Объект "появляется на свет" в результате вызова специального метода, который инициализирует объект – *конструктора.* Созданный экземпляр уничтожается другим методом – *деструктором:*

```
AMyObject := TMyObject.Create;
{ действия с созданным объектом }
. . . . . .
AMyObject.Destroy;
```

Но ведь объекта еще нет, как мы можем вызывать его методы? Однако обратите внимание, что вызывается метод TMyObject.Create, а не AMyObject.Create. Есть такие методы (в том числе конструктор), которые

успешно работают до (или даже *без)* создания объекта. О подобных методах, называемых *методами класса,* пойдет речь чуть ниже.

В Object Pascal конструкторов у класса может быть несколько. Общепринято называть конструктор create (в отличие от Turbo Pascal, где конструктор обычно назывался init, и от C++, где его имя совпадает с именем класса).

Типичное название деструктора – Destroy.

```
type
   TMyObject = class(TObject)
      MyField: Integer;
      Constructor Create;
      Destructor Destroy;
      Function MyMethod: Integer;
   end;
```

Для уничтожения экземпляра объекта рекомендуется использовать метод Free, который первоначально проверяет указатель (не равен ли он **nil**) и только затем вызывает Destroy:

```
AMyObject.Free;
```

До передачи управления телу конструктора происходит собственно создание объекта – под него отводится память, значения всех полей обнуляются. Далее выполняется код конструктора, написанный программистом для инициализации экземпляров данного класса. Таким образом, хотя на первый взгляд синтаксис конструктора схож с вызовом процедуры (не определено возвращаемое значение), но на самом деле конструктор – это функция, возвращающая созданный и инициализированный объект.

Примечание

Конструктор создает новый объект только в том случае, если перед его именем указано имя класса. Если указать имя уже существующего объекта, он поведет себя по-другому: не создаст новый объект, а только выполнит код, содержащийся в теле конструктора.

Чтобы правильно инициализировать в создаваемом объекте поля, относящиеся к классу-предку, нужно сразу же при входе в конструктор вызвать конструктор предка при помощи зарезервированного слова inherited:

```
constructor TMyObject.Create;
begin
   inherited Create;
   . . . . .
end;
```

Взяв любой из примеров, поставляемых вместе в Delphi, вы почти не увидите там вызовов конструкторов и деструкторов. Дело в том, что любой компонент, попавший при визуальном проектировании в ваше приложение из Палитры компонентов, включается в определенную иерархию. Иерархия эта замыкается на форме (класс TForm): для всех ее составных частей

конструкторы и деструкторы вызываются автоматически, незримо для программиста. Кто создает и уничтожает формы? Это делает приложение (глобальный объект с именем Application). В файле проекта, (с расширением dpr) вы можете увидеть вызовы метода Application.CreateForm, предназначенного для этой цели.

Что же касается объектов, создаваемых динамически (во время выполнения приложения), то здесь нужен явный вызов конструктора и метода Free.

Поля, свойства и методы

Поля класса являются переменными, объявленными внутри класса. Они предназначены для хранения данных во время работы экземпляра класса (объекта). Ограничений на тип полей в классе не предусмотрено. В описании класса поля должны предшествовать методам и свойствам. Обычно поля используются для обеспечения выполнения операций внутри класса.

Примечание

При объявлении имен полей принято к названию добавлять заглавную букву F. Например FSomeField.

Итак, поля предназначены для использования внутри класса. Однако класс должен каким-либо образом взаимодействовать с другими классами или программными элементами приложения. В подавляющем большинстве случаев класс должен выполнить с некоторыми данными определенные действия и представить результат.

Для получения и передачи данных в классе применяются свойства. Для объявления свойств в классе используется зарезервированное слово property.

Свойства представляют собой атрибуты, которые составляют индивидуальность объекта и помогают описать его. Например, обычная кнопка в окне приложения обладает такими свойствами, как цвет, размеры, положение.

Для экземпляра класса "кнопка" значения этих атрибутов задаются при помощи свойств – специальных переменных, определяемых ключевым словом property. Цвет может задаваться свойством color, размеры – свойствами Width, Height и т. д.

Так как свойство обеспечивает обмен данными с внешней средой, то для доступа к его значению используются специальные методы класса. Поэтому обычно свойство определяется тремя элементами: полем и двумя методами, которые осуществляют его чтение/запись:

```
type
   TAnObject = class(TObject)
      function GetColor: TSomeType;
      procedure SetColor(ANewValue: TSomeType);
      property AColor: TSomeType read GetColor write SetColor;
   end;
```

В данном примере доступ к значению свойства AColor осуществляется через вызовы методов GetColor и SetColor. Однако в обращении к этим методам в явном виде нет необходимости: достаточно написать:

```
AnObject.AColor := AValue;
AVariable := AnObject.AColor;
```

и компилятор самостоятельно оттранслирует обращение к свойству AColor в вызовы методов GetColor или SetColor. То есть внешне свойство выглядит в точности как обычное поле, но за всяким обращением к нему могут стоять нужные вам действия. Например, если у вас есть объект, представляющий собой квадрат на экране, и его свойству "цвет" вы присваиваете значение "белый", то произойдет немедленная перерисовка, приводящая реальный цвет на экране в соответствие со значением свойства. Выполнение этой операции осуществляется методом, который связан с установкой значения свойства "цвет".

В методах, входящих в состав свойств, может осуществляться проверка устанавливаемой величины на попадание в допустимый диапазон значений и вызов других процедур, зависящих от вносимых изменений. Если же потребности в специальных процедурах чтения и/или записи нет, можно вместо имен методов применять имена полей. Рассмотрим следующую конструкцию:

```
TPropObject = class(TObject)
   FValue: TSomeType;
   procedure DoSomething;
   function Correct(AValue: Integer):boolean;
   procedure SetValue(NewValue: Integer);
   property AValue: Integer read FValue write SetValue;
end;

procedure TPropObject.SetValue(NewValue: Integer);
begin
   if (NewValue <> FValue) and Correct(NewValue) then
     FValue := NewValue;
   DoSomething;
end;
```

В этом примере чтение значения свойства AValue означает просто чтение поля FValue. Зато при присвоении значения внутри SetValue вызывается сразу два метода.

Если свойство должно только читаться или записываться, в его описании может присутствовать соответствующий метод:

```
type
   TAnObject = class(TObject)
      property AProperty: TSomeType read GetValue;
   end;
```

В этом примере вне объекта значение свойства можно лишь прочитать; попытка присвоить свойству AProperty значение вызовет ошибку компиляции.

Для присвоения свойству значения по умолчанию используется ключевое слово default:

```
property Visible: boolean read FVisible write SetVisible
default True;
```

Это означает, что при запуске программы свойство будет установлено компилятором в True.

Свойство может быть и векторным; в этом случае оно внешне выглядит как массив:

```
property APoints[Index : Integer]:TPoint read GetPoint write
SetPoint;
```

На самом деле в классе может и не быть соответствующего поля – массива. Напомним, что вся обработка обращений к внутренним структурам класса может быть замаскирована.

Для векторного свойства необходимо описать не только тип элементов массива, но также имя и тип индекса. После ключевых слов read и write в этом случае должны стоять имена методов – использование здесь полей массивов недопустимо. Метод, читающий значение векторного свойства, должен быть описан как функция, возвращающая значение того же типа, что и элементы свойства, и имеющая единственный параметр того же типа и с тем же именем, что и индекс свойства:

```
function GetPoint(Index:Integer):TPoint;
```

Аналогично, метод, помещающий значения в такое свойство, должен первым параметром иметь индекс, а вторым – переменную нужного типа (которая может быть передана как по ссылке, так и по значению):

```
procedure SetPoint(Index:Integer; NewPoint:TPoint);
```

У векторных свойств есть еще одна важная особенность. Некоторые классы в Delphi (списки TList, наборы строк TStrings) "построены" вокруг основного векторного свойства. Основной метод такого класса дает доступ к некоторому массиву, а все остальные методы являются как бы вспомогательными. Специально для облегчения работы в этом случае векторное свойство может быть описано с ключевым словом default:

```
type
   TMyObject = class;
      property Strings[Index: Integer]: string read Get
            write Put; default;
   end;
```

Если у объекта есть такое свойство, то можно его не упоминать, а ставить индекс в квадратных скобках сразу после имени объекта:

```
var AMyObject: TMyObject;
begin
   AMyObject.Strings[1] := 'First' ; {первый способ}
   AMyObject[2] := 'Second'; {второй способ}
end
```

Будьте внимательны, применяя зарезервированное слово default, – как мы увидели, для обычных и векторных свойств оно употребляется в разных случаях и с различным синтаксисом.

О роли свойств в Delphi красноречиво говорит следующий факт: у всех имеющихся в распоряжении программиста стандартных классов 100% полей недоступны и заменены базирующимися на них свойствами. Рекомендуем при разработке собственных классов придерживаться этого же правила.

Обратим внимание на то, что при объяснении терминов "поле" и "свойство" мы использовали понятие метода. Итак, методом называется объявленная в классе функция или процедура, которая используется для работы с полями и свойствами класса. Согласно принципам ООП *(см. разд. "Инкапсуляция" далее в этой главе),* обращаться к свойствам класса можно только через его методы. От обычных процедур и функций методы отличаются тем, что им при вызове передается указатель на тот объект, который их вызвал. Поэтому обрабатываться будут данные именно того объекта, который вызвал метод. На некоторых особенностях использования методов мы остановимся ниже.

События

Программистам, давно работающим в Windows, наверное, не нужно пояснять смысл термина "событие". Сама эта среда и написанные для нее программы управляются событиями, возникающими в результате воздействий пользователя, аппаратуры компьютера или других программ. Весточка о наступлении события – это сообщение Windows, полученное так называемой функцией окна. Таких сообщений сотни, и, по большому счету, написать программу для Windows – значит определить и описать реакцию на некоторые из них.

Работать с таким количеством сообщений, даже имея под рукой справочник, нелегко. Поэтому одним из больших преимуществ Delphi является то, что программист избавлен от необходимости работать с сообщениями Windows (хотя такая возможность у него есть). Типовых событий в Delphi – не более двух десятков, и все они имеют простую интерпретацию, не требующую глубоких знаний среды.

Рассмотрим, как реализованы события на уровне языка Object Pascal. *Событие* – это свойство процедурного типа, предназначенное для создания пользовательской реакции на то или иное входное воздействие:

```
property OnMyEvent: TMyEvent read FOnMyEvent write FOnMyEvent;
```

Здесь FOnMyEvent – поле процедурного типа, содержащее адрес некоторого метода. Присвоить такому свойству значение – значит указать объекту адрес метода, который будет вызываться в момент наступления события. Такие методы называют обработчиками событий. Например, запись:

```
Application.OnActivate := MyActivatingMethod;
```

означает, что при активизации объекта Application (так называется объект, соответствующий работающему приложению) будет вызван метод-обработчик MyActivatingMethod.

Внутри библиотеки времени выполнения Delphi вызовы обработчиков событий находятся в методах, обрабатывающих сообщения Windows. Выполнив необходимые действия, этот метод проверяет, известен ли адрес обработчика, и, если это так, вызывает его:

```
if Assigned(FOnMyEvent) then FOnMyEvent(Self);
```

События имеют разное количество и тип параметров в зависимости от происхождения и предназначения. Общим для всех является параметр `Sender` – он указывает на объект-источник события. Самый простой тип – TNotifyEvent – не имеет других параметров:

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

Тип метода, предназначенный для извещения о нажатии клавиши, предусматривает передачу программисту кода этой клавиши о передвижении мыши – ее текущих координат и т. п. Все события в Delphi принято предварять префиксом on: onCreate, OnMouseMove, onPaint и т. д. Дважды щелкнув в Инспекторе объектов на странице Events в поле любого события, вы получите в программе заготовку метода нужного типа. При этом его имя будет состоять из имени текущего компонента и имени события (без префикса on), а относиться он будет к текущей форме. Пусть, например, на форме Form1 есть текст Label1. Тогда для обработки щелчка мышью (событие OnClick) будет создана заготовка метода TForm1.Label1Click:

```
procedure TForm1.Label1Click(Sender: TObject);
begin

end;
```

Поскольку события – это свойства объекта, их значения можно изменять в любой момент во время выполнения программы. Эта замечательная возможность называется *делегированием.* Можно в любой момент взять способы реакции на события у одного объекта и присвоить (делегировать) их другому:

```
Object1.OnMouseMove := Object2.OnMouseMove;
```

Принцип делегирования позволяет избежать трудоемкого процесса порождения новых дочерних классов для каждого специфического случая, заменяя его простой подстановкой процедур. При необходимости можно выбирать один из нескольких возможных вариантов обработчиков событий.

Но какой механизм позволяет подменять обработчики, ведь это не просто процедуры, а методы? Здесь как нельзя кстати приходится существующее в Object Pascal понятие указателя на метод. Отличие метода от процедуры состоит в том, что помимо явно описанных параметров методу всегда неявно передается еще и указатель на вызвавший его экземпляр класса (переменная Self). Вы можете описать процедурный тип, который будет

совместим по присваиванию с методом (т. е. предусматривать получение Self). Для этого в описание процедуры нужно добавить зарезервированные слова of object. Указатель на метод – это указатель на такую процедуру.

```
type
   TMyEvent = procedure(Sender: TObject; var AValue: Integer)
               of object;
   T1stObject = class;
      FOnMyEvent: TMyEvent;
      property OnMyEvent: TMyEvent read FOnMyEvent
          write FOnMyEvent;
   end;

   T2ndObject = class;
      procedure SetValue1(Sender: TObject;
         var AValue:  Integer);
      procedure SetValue2(Sender: TObject;
         var AValue: Integer);
   end;

var
   Obj1: T1stObject;
   Obj2: T2ndObject;
begin
   Obj1 := T1stObject.Create;
   Obj2 := T2ndObject.Create;
   Obj1.OnMyEvent := Obj2.SetValue1;
   Obj1.OnMyEvent := Obj2.SetValue2;
end;
```

Этот пример показывает, что при делегировании можно присваивать методы других классов. Здесь обработчиком события OnMyEvent объекта Obj1 по очереди выступают методы SetValue1 и SetValue2 объекта Obj2.

Обработчики событий нельзя сделать просто процедурами – *они обязательно должны быть чьими-то методами.* Но их можно "отдать" какому-либо другому объекту. Более того, для этих целей можно описать и создать специальный объект. Его единственное предназначение – быть носителем методов, которые затем делегируются другим объектам. Разумеется, такой объект надо не забыть создать до использования его методов, а в конце – уничтожить. Можно и не делать этого, объявив методы методами класса, о которых речь пойдет в одном из последующих разделов.

Мы сейчас решили задачу использования нескольких разных обработчиков того или иного события для одного объекта. Но не менее часто требуется решить обратную задачу – а как использовать для различных событий разных объектов один и тот же обработчик?

Если никакой "персонификации" объекта, вызвавшего метод, не нужно, все делается тривиально и проблемы не возникает. Самый простой пример: в современных программах основные функции дублируются дважды – в меню и на панели инструментов. Естественно, сначала нужно создать и наполнить

метод содержимым (скажем, для пункта меню), а затем в Инспекторе объектов указать его же для кнопки панели инструментов.

Более сложный случай, когда внутри такого метода нужно разобраться, кто собственно его вызвал. Если потенциальные кандидаты имеют разный объектный тип (как в предыдущем абзаце – кнопка и пункт меню), то именно объектный тип можно применить в качестве критерия:

```
If Sender is TMenuItem then ShowMessage('Выбран пункт меню');
```

Если же все объекты, разделяющие между собой один обработчик события, относятся к одному классу, то приходится прибегать к дополнительным ухищрениям. Типовой приём – использовать свойство Tag, которое имеется у всех компонентов, и, вполне вероятно, именно для этого и задумывалось:

```
const
   colors : array[0..7] of TColor =
    (clWhite, clRed, clBlue, clYellow, clAqua, clGreen,
     clMaroon, clBlack);

procedure TForml.CheckBoxClick(Sender: TObject);
begin
   with TCheckBox(Sender) do
      if Checked then Color := Colors[Tag]
      else Color := clBtnFace;
end;
```

Пусть в форме имеется несколько переключателей. Для того чтобы при нажатии каждый из них окрашивался в свой цвет, нужно в Инспекторе объектов присвоить свойству Tag значения от 0 до 7 и для каждого связать событие OnClick с методом CheckBoxClick. Этот единственный метод справится с задачей для всех переключателей.

Инкапсуляция

В предыдущих разделах мы ввели ряд новых понятий, которыми будем пользоваться в дальнейшем. Теперь поговорим о принципах, составляющих суть объектно-ориентированного программирования. Таких принципов три – *инкапсуляция, наследование и полиморфизм*.

Как правило, объект – это сложная конструкция, свойства и поведение составных частей которой находятся во взаимодействии. К примеру, если мы моделируем взлетающий самолет, то после набора им определенной скорости и отрыва от земли принципы управления им полностью изменяются. Поэтому в объекте "самолет" явно недостаточно просто увеличить значение поля "скорость" на несколько километров в час – такое изменение должно автоматически повлечь за собой целый ряд других.

При создании программных объектов подобные ситуации можно моделировать, связывая со свойствами необходимые методы. Понятие инкапсуляции соответствует этому механизму.

Классическое правило объектно-ориентированного программирования утверждает, что для обеспечения надежности нежелателен прямой доступ к полям объекта: чтение и обновление их содержимого должно производиться посредством вызова соответствующих методов. Это правило и называется *инкапсуляцией.* В старых реализациях ООП (например, в Turbo Pascal) эта мысль внедрялась только посредством призывов и примеров в документации; в языке же Object Pascal есть соответствующая конструкция. В Delphi пользователь вашего объекта может быть полностью отгорожен от полей при помощи свойств *(см. выше).*

Примечание

Дополнительные возможности ограничения доступа к нужным данным обеспечивают области видимости *(см. ниже).*

Наследование

Вторым "столпом" ООП является *наследование.* Этот простой принцип означает, что если вы хотите создать новый класс, лишь немного отличающийся от старого, то совершенно нет необходимости в переписывании заново уже существующих полей и методов. Вы объявляете, что новый класс TNewObject

```
TNewObject = class(TOldObject);
```

является *потомком* или *дочерним классом* старого класса TOldObject, называемого *предком* или *родительским классом,* и добавляете к нему новые поля, методы и свойства – иными словами, то, что нужно при переходе от общего к частному.

Примечание

Прекрасный пример, иллюстрирующий наследование, представляет собой иерархия классов VCL.

В Object Pascal все классы являются потомками класса TObject. Поэтому, если вы создаете дочерний класс прямо от TObject, то в определении его можно не упоминать. Следующие два выражения одинаково верны:

```
TMyObject = class(TObject);
TMyObject = class;
```

Первый вариант, хотя он и более длинный, предпочтительнее – для устранения возможных неоднозначностей. Класс TObject несет очень серьезную нагрузку и будет рассмотрен отдельно.

Унаследованные от класса-предка поля и методы доступны в дочернем классе; если имеет место совпадение имен методов, то говорят, что они *перекрываются*.

Поведение методов при наследовании, без преувеличения, является краеугольным камнем объектно-ориентированного программирования. В зависимости от того, какие действия происходят при вызове, методы делятся

на три группы. В *первую* группу отнесем статические методы, во *вторую* – виртуальные (virtual) и динамические (dynamic) и, наконец, в *третью* – перегружаемые (overload) методы.

Методы первой группы полностью перекрываются в классах-потомках при их переопределении. При этом можно полностью изменить объявление метода. Методы второй группы при наследовании должны сохранять наименование и тип. Перегружаемые методы дополняют механизм наследования возможностью использовать нужный вариант метода (собственный или родительский) в зависимости от условий применения. Подробно все эти методы обсуждаются ниже.

Язык C++ допускает так называемое множественное наследование. В этом случае новый класс может наследовать часть своих элементов от одного родительского класса, а часть – от другого. Это, наряду с удобствами, зачастую приводит к проблемам. В Object Pascal понятие множественного наследования отсутствует. Если вы хотите, чтобы новый класс объединял свойства нескольких, породите классы-предки один от другого или включите в один класс несколько полей, соответствующих другим желаемым классам.

Полиморфизм

Рассмотрим внимательно следующий пример. Пусть у нас имеются некое обобщенное поле для хранения данных – класс TField и три его потомка – для хранения строк, целых и вещественных чисел:

```
type
   TField = class
      function GetData:string; virtual; abstract;
   end;
   TStringField = class(TField)
      FData : string;
      function GetData: string; override;
   end;
   TIntegerField = class(TField)
      FData : Integer;
      function GetData: string; override;
   end;
   TExtendedField = class(TField)
      FData : Extended;
      function GetData: string; override;
   end;
function TStringField.GetData;
begin
   Result := FData;
end;
function TIntegerField.GetData;
begin
   Result := IntToStr(FData);
end;
```

```
function TExtendedField.GetData;
begin
   Result:= FloatToStrF(FData, ffFixed, 7, 2);
end;

procedure ShowData(AField : TField);
begin
   Form1.Label1.Caption := AField.GetData;
end;
```

В этом примере классы содержат разнотипные поля данных FData и только то и "умеют", что сообщить о значении этих данных текстовой строкой (при помощи Метода GetData). Внешняя по отношению к ним процедура ShowData получает объект в виде параметра и показывает эту строку.

Правила контроля *соответствия типов* (typecasting) языка Object Pascal гласят, что *объекту как указателю на экземпляр объектного типа может быть присвоен адрес любого экземпляра любого из дочерних типов.* В процедуре ShowData параметр описан как TField – это значит, что в нее можно передавать объекты классов и TStringField, и TIntegerField, и TExtendedField, и любого другого потомка класса TField.

Но какой (точнее, чей) метод GetData при этом будет вызван? Тот, который соответствует классу *фактически* переданного объекта. Этот принцип называется *полиморфизмом,* и он, пожалуй, представляет собой наиболее важный козырь ООП.

Допустим, вы имеете дело с некоторой совокупностью явлений или процессов. Чтобы смоделировать их средствами ООП, нужно выделить их самые общие, типовые черты. Те из них, которые не изменяют своего содержания, должны быть реализованы в виде статических методов. Те же, которые изменяются при переходе от общего к частному, лучше облечь в форму виртуальных методов.

Основные, "родовые" черты (методы) нужно описать в классе-предке и затем перекрывать их в классах-потомках. В нашем примере программисту, пишущему процедуру вроде ShowData, важно лишь, что любой объект, переданный в нее, является потомком TField и он умеет сообщить о значении своих данных (выполнив метод GetData). Если, к примеру, такую процедуру скомпилировать и поместить в динамическую библиотеку, то эту библиотеку можно будет раз и навсегда использовать без изменений, хотя будут появляться и новые, неизвестные в момент ее создания классы-потомки TField.

Наглядный пример использования полиморфизма дает среда Delphi. В ней имеется класс TComponent, на уровне которого сосредоточены определенные "правила" взаимодействия компонентов со средой разработки и с другими компонентами. Следуя этим правилам, можно порождать от класса TComponent свои компоненты, настраивая Delphi на решение специальных задач.

Методы

Из предыдущего материала вы узнали, что функционирование объектов обеспечивается различными типами методов, которые различаются особенностями реализации механизма наследования. Теперь рассмотрим эти методы более подробно.

Абстрактными называются методы, которые определены в классе, но не содержат никаких действий, никогда не вызываются и обязательно должны быть переопределены в потомках класса. Абстрактными могут быть только виртуальные и динамические методы. В Object Pascal такие методы объявляются с помощью одноименной директивы. Она указывается при описании метода:

```
procedure NeverCallMe; virtual; abstract;
```

При этом никакого кода для этого метода писать не нужно. Вызов метода NeverCallMe приведет к созданию исключительной ситуации EAbstractError.

Пример с классом TField из *разд. "Полиморфизм" этой главы* поясняет, для чего нужно использование абстрактных методов. В данном случае класс TField не используется сам по себе; его основное предназначение – быть родоначальником иерархии конкретных классов-"полей" и дать возможность абстрагироваться от частностей. Хотя параметр процедуры ShowData и описан как TField, но, если передать в нее объект этого класса, произойдет исключительная ситуация вызова абстрактного метода.

Статические методы, а также любые поля в объектах-потомках ведут себя одинаково: вы можете без ограничений перекрывать старые имена и при этом изменять тип методов. Код нового статического метода полностью перекрывает (заменяет собой) код старого метода:

```
type
   T1stObj = class
      i : Extended;
      procedure SetData(AValue: Extended);
   end;
   T2ndObj = class(T1stObj)
      i : Integer;
      procedure SetData(AValue: Integer);
   end;
```

Примечание

В практике программирования принято присваивать всем идентификаторам в программе (в том числе полям объектов) осмысленные названия. Это существенно облегчит работу с исходным кодом не только другим разработчикам, но и вам.

В отличие от поля, внутри других методов перекрытый метод доступен при указании зарезервированного слова inherited. По умолчанию все методы объектов являются статическими – их адрес определяется еще на стадии компиляции проекта, поэтому они вызываются быстрее всего. Может быть,

вам еще не ясно, для чего упоминается этот факт. Просто запомните его, он понадобится при сравнении статических и виртуальных методов.

Принципиально отличаются от статических *виртуальные* и *динамические* методы. Они должны быть объявлены путем добавления соответствующей директивы virtual или dynamic. С точки зрения наследования методы этих двух видов одинаковы: они могут быть перекрыты в дочернем классе только одноименными методами, имеющими тот же тип.

Если задуматься над рассмотренным выше примером, становится ясно, что у компилятора нет возможности определить класс объекта, фактически переданного в процедуру ShowData. Нужен механизм, позволяющий определить это прямо во время выполнения. Такой механизм называется *поздним связыванием* (late binding).

Естественно, что этот механизм должен быть каким-то образом связан с передаваемым объектом. Для этого используются *таблица виртуальных методов* (Virtual Method Table, VMT) и *таблица динамических методов* (Dynamic Method Table, DMT).

Разница между виртуальными и динамическими методами заключается в особенности поиска адреса. Когда компилятор встречает обращение к виртуальному методу, он подставляет вместо прямого вызова по конкретному адресу код, который обращается к VMT и извлекает оттуда нужный адрес.

Такая таблица есть для каждого класса (объектного типа). В ней хранятся адреса *всех* виртуальных методов класса, независимо от того, унаследованы ли они от предка или перекрыты в данном классе. Отсюда и достоинства, и недостатки виртуальных методов: они вызываются сравнительно быстро, однако для хранения указателей на них в таблице VMT требуется большое количество памяти.

Динамические методы вызываются медленнее, но позволяют более экономно расходовать память. Каждому динамическому методу системой присваивается уникальный индекс. В таблице динамических методов класса хранятся индексы и адреса *только тех* динамических методов, которые описаны в данном классе. При вызове динамического метода происходит поиск в этой таблице; в случае неудачи просматриваются таблицы DMT всех классов-предков в порядке иерархии и, наконец, класс TObject, где имеется стандартный обработчик вызова динамических методов. Экономия памяти налицо.

Для перекрытия и виртуальных, и динамических методов служит директива override, с помощью которой (и только с ней!) можно переопределять оба этих типа методов. Приведем пример:

```
type
   TFirstClass = class
      FMyField1: Integer;
      FMyField2: Longint;
      procedure StatMethod;
      procedure VirtMethod1; virtual;
```

```
      procedure VirtMethod2; virtual;
      procedure DynaMethod1; dynamic
      procedure DynaMethod2; dynamic
   end;
   TSecondClass = class(TFirstClass)
      procedure StatMethod;
      procedure VirtMethodl; override;
      procedure DynaMethodl; override;
   end;
var
   Obj1: TFirstClass;
   Obj2: TSecondClass;
```

Первый из методов в примере создается заново, остальные два – перекрываются. Попытка применить директиву override к статическому методу вызовет ошибку компиляции.

Примечание

Будьте внимательны: попытка перекрытия с директивой не override, а virtual или dynamic приведет на самом деле к созданию нового одноименного метода.

Перегрузка методов

Есть еще одна, совершенно особенная разновидность методов – *перегружаемые.*

Эту категорию методов нельзя назвать антагонистом двух предыдущих: и статические, и виртуальные, и динамические методы могут быть перегружаемыми. *Перегрузка методов нужна, чтобы произвести одинаковые или похожие действия с разнотипными данными.*

Рассмотрим немного измененный пример, иллюстрирующий статические методы:

```
type
   T1stObj = class
      FExtData : Extended;
      procedure SetData(AValue: Extended);
   end;
   T2ndObj = class(T1stObj)
      FIntData : Integer;
      procedure SetData(AValue: Integer);
   end;
var
   T1: T1stObj;
   T2: T2ndObj;
```

В этом случае попытка вызова из объекта T2 методов

```
T2.SetData(1.0);
T2.SetData(1);
```

вызовет ошибку компиляции на первой из двух строк. Для компилятора внутри T2 статический метод с параметром типа extended перекрыт, и он его

"не признает". Где же выход из сложившегося положения? Переименовать один из методов, например создать SetIntegerData и SetExtendedData?

Можно, но если методов не два, а, скажем, сто, моментально возникнет путаница. Сделать методы виртуальными? Нельзя, поскольку тип и количество параметров в одноименных виртуальных методах должны в точности совпадать. Теперь для этого существуют перегружаемые методы, объявляемые при помощи директивы overload:

```
type
   T1stObj = class
      FExtData : Extended;
      procedure SetData(AValue: Extended); overload;
   end;

   T2ndObj = class(T1stObj)
      FIntData : Integer;
      procedure SetData(AValue: Integer); overload;
   end;
```

Объявив метод SetData перегружаемым, в программе можно использовать обе его реализации одновременно. Это возможно потому, что компилятор определяет тип передаваемого параметра (целый или с плавающей точкой) и в зависимости от этого подставит вызов соответствующего метода: для целочисленных данных – метод объекта T2ndObj, для данных с плавающей точкой – метод объекта T1stObj.

Можно перегрузить и виртуальный (динамический) метод. Надо только в этом случае добавить директиву reintroduce:

```
type
   T1stObj = class
      FExtData : Extended;
      procedure SetData(AValue: Extended); overload; virtual;
   end;

   T2ndObj = class(T1stObj)
      FIntData : Integer;
      procedure SetData(AValue: Integer); reintroduce; overload;
   end;
```

На перегрузку методов накладывается ограничение – нельзя перегружать методы, находящиеся в области видимости published, *т. е. те, которые будут использоваться в Инспекторе объектов.*

Области видимости

При описании нового класса важен разумный компромисс. С одной стороны, требуется скрыть от других методы и поля, представляющие собой внутреннее устройство класса (для этого и придуманы свойства). Маловажные детали на уровне пользователя объекта будут бесполезны и только помешают целостности восприятия.

С другой стороны, если слишком ограничить того, кто будет порождать классы-потомки, и не обеспечить ему достаточный набор инструментальных средств и свободу маневра, то он и не станет использовать ваш класс.

В модели объектов языка Object Pascal существует механизм доступа к составным частям объекта, определяющий области, где ими можно пользоваться *(области видимости)*. Поля и методы могут относиться к четырем группам (секциям), отличающимся областями видимости. Методы и свойства могут быть *общими* (секция public), *личными* (секция private), *защищёнными* (секция protected) и *опубликованными* (секция published).

Области видимости, определяемые первыми тремя директивами, таковы.

- Поля, свойства и методы секции public не имеют ограничений на видимость. Они доступны из других функций и методов объектов как в данном модуле, так и во всех прочих, ссылающихся на него.
- Поля, свойства и методы, находящиеся в секции private, доступны только в методах класса и в функциях, содержащихся в том же модуле, что и описываемый класс. Такая директива позволяет полностью скрыть детали внутренней реализации класса. Свойства и методы из секции private можно изменять, и это не будет сказываться на программах, работающих с объектами этого класса. Единственный способ для кого-то другого обратиться к ним – переписать заново созданный вами модуль (если, конечно, доступны исходные тексты).
- Поля, свойства и методы секции protected также доступны только внутри модуля с описываемым классом. Но – и это главное – они доступны в классах, являющихся потомками данного класса, в том числе и в других модулях. Такие элементы особенно необходимы для разработчиков новых компонентов – потомков уже существующих. Оставляя свободу модернизации класса, они все же скрывают детали реализации от того, кто только пользуется объектами этого класса.

Рассмотрим пример, иллюстрирующий три варианта областей видимости.

```
unit First;

interface

type
   TFirstObj = class
   private
      procedure Method1;
   protected
      procedure Method2;
   public
      procedure Method3;
   end;

procedure TestProc1;
```

```
implementation

uses dialogs;

var AFirstObj: TFirstObj;

procedure TestProc1;
begin
   AFirstObj := TFirstObj.Create;
   AFirstObj.Method1; {допустимо}
   AFirstObj.Method2; {допустимо}
   AFirstObj.Method3; {допустимо}
   AFirstObj.Free;
end;

procedure TFirstObj.Method1;
begin
   ShowMessage('1');
end;

procedure TFirstObj.Method2;
begin
   ShowMessage('2');
   Method1;
end;

procedure TFirstObj.Method3;
begin
   ShowMessage('3');
   Method2;
end;


end.
```

```
unit Second;

interface

uses First;

type
   TSecondObj = class(TFirstObj)
      procedure Method4;
   end;

procedure TestProc2;


implementation

var
   AFirstObj : TFirstObj;
   ASecondObj: TSecondObj;
```

```
procedure TSecondObj.Method4;
begin
   Method1; {недопустимо – произойдет ошибка компиляции}
   Method2; {допустимо}
   Method3; {допустимо}
end;

procedure TestProc2;
begin
   AFirstObj := TFirstObj.Create;
   AFirstObj.Method1; {недопустимо}
   AFirstObj.Method2; {недопустимо}
   AFirstObj.Method3; {допустимо}
   AFirstObj.Free;
   ASecondObj := TSecondObj.Create;
   ASecondObj.Method1; {недопустимо}
   ASecondObj.Method2; {допустимо}
   ASecondObj.Method3; {допустимо}
   ASecondObj.Free;
end;

end.
```

Если к этому примеру добавить модуль Third и попробовать вызвать методы классов TFirstObj и TSecondObj оттуда, то к числу недоступных будет отнесен и Method2 – он доступен только в том модуле, в котором описан.

Наконец, область видимости, определяемая четвертой директивой – published, имеет особое значение для интерфейса визуального проектирования Delphi. В этой секции должны быть собраны те свойства объекта, которые будут видны не только во время исполнения приложения, но и из среды разработки. Публиковать можно свойства большинства типов, за исключением старого типа real (теперь он называется real48), свойств типа "массив" и некоторых других. Все свойства компонентов, доступные через Инспектор объектов, являются их опубликованными свойствами. Во время выполнения такие свойства общедоступны, как и public.

Три области видимости – private, protected, public – как бы упорядочены по возрастанию видимости методов. В классах-потомках можно повысить видимость методов и свойств, но не понизить ее. При описании дочернего класса можно переносить методы и свойства из одной сферы видимости в другую, не переписывая их заново и даже не описывая – достаточно упомянуть о нем в другом месте:

```
type
   TFirstObj = class
   private
      FNumber: Integer;
   protected
      property Number: Integer read FNumber;
   end;

   TSecondObj = class(TFirstObj)
```

```
   published
      property Number;
   end;
```

Если какое-либо свойство объекта из состава VCL принадлежит к области public, вернуть его в private невозможно. Напротив, обратная процедура широко практикуется в Delphi. У многих компонентов (например, TEdit) есть предок (в данном случае TCustomEdit), который отличается только отсутствием опубликованных свойств. Так что, если вы хотите создать новый редактирующий компонент, порождайте его на базе TCustomEdit и публикуйте только те свойства, которые считаете нужными. Разумеется, если вы поместили свойство в область private, "достать" его оттуда в потомках возможности уже нет.

Объект изнутри

Теперь, когда мы разобрались с основными определениями и механизмами ООП, настало время более подробно изучить, что представляет собой объект и как он работает. Ясно, что каждый экземпляр класса содержит отдельную копию всех его полей. Ясно, что где-то в его недрах есть указатели на таблицу виртуальных методов и таблицу динамических методов. А что еще там имеется? И как происходит вызов методов? Вернёмся к примеру из *разд. "Полиморфизм" данной главы:*

```
type
   TFirstClass = class
      FMyField1: Integer;
      FMyField2: Longint;
      procedure StatMethod;
      procedure VirtMethod1; virtual;
      procedure VirtMethod2; virtual;
      procedure DynaMethod1; dynamic;
      procedure DynaMethod2; dynamic;
   end;

   TSecondClass = class(TFirstClass)
      procedure StatMethod;
      procedure VirtMethod1; override;
      procedure DynaMethod1; override;
   end;

   Obj1: TFirstClass;
   Obj2: TSecondClass;
```

На рис. 1.1 показано, как будет выглядеть внутренняя структура рассмотренных в нем объектов.

Первое поле каждого экземпляра того или иного объекта содержит указатель на его класс. Класс как структура состоит из двух частей. Начиная с адреса, на который ссылается указатель на класс, располагается таблица виртуальных методов. Напомним, что она содержит адреса всех виртуальных методов класса, включая унаследованные от предков. Длина таблиц VMT

объектов Obj1 и Obj2 одинакова – по два элемента (8 байт). Перед таблицей виртуальных методов расположена специальная структура, содержащая дополнительную служебную информацию. В ней содержатся данные, полностью характеризующие класс: его имя, размер экземпляра, указатели на класс-предок, имя класса и т. д. На рис. 1.1 она показана одним блоком, а ее содержимое расшифровано ниже.

Одно из полей структуры содержит адрес таблицы динамических методов класса (DMT). Таблица имеет следующий формат – в начале слово, содержащее количество элементов таблицы; затем – слова, соответствующие индексам методов. Нумерация индексов начинается с -1 и идет по убывающей. После индексов идут собственно адреса динамических методов. Обратите внимание, что DMT объекта Obj1 состоит из двух элементов, Obj2 – из одного, соответствующего перекрытому методу DynaMethod1. В случае вызова Obj2. DynaMethod2 индекс не будет найден в таблице DMT Obj2, и произойдет обращение к DMT Obj1. Именно так экономится память при использовании динамических методов.

В языке Object Pascal определены два оператора – is и as, неявно обращающиеся к таблице динамических методов. Оператор is предназначен для проверки совместимости по присваиванию экземпляра объекта с заданным классом.

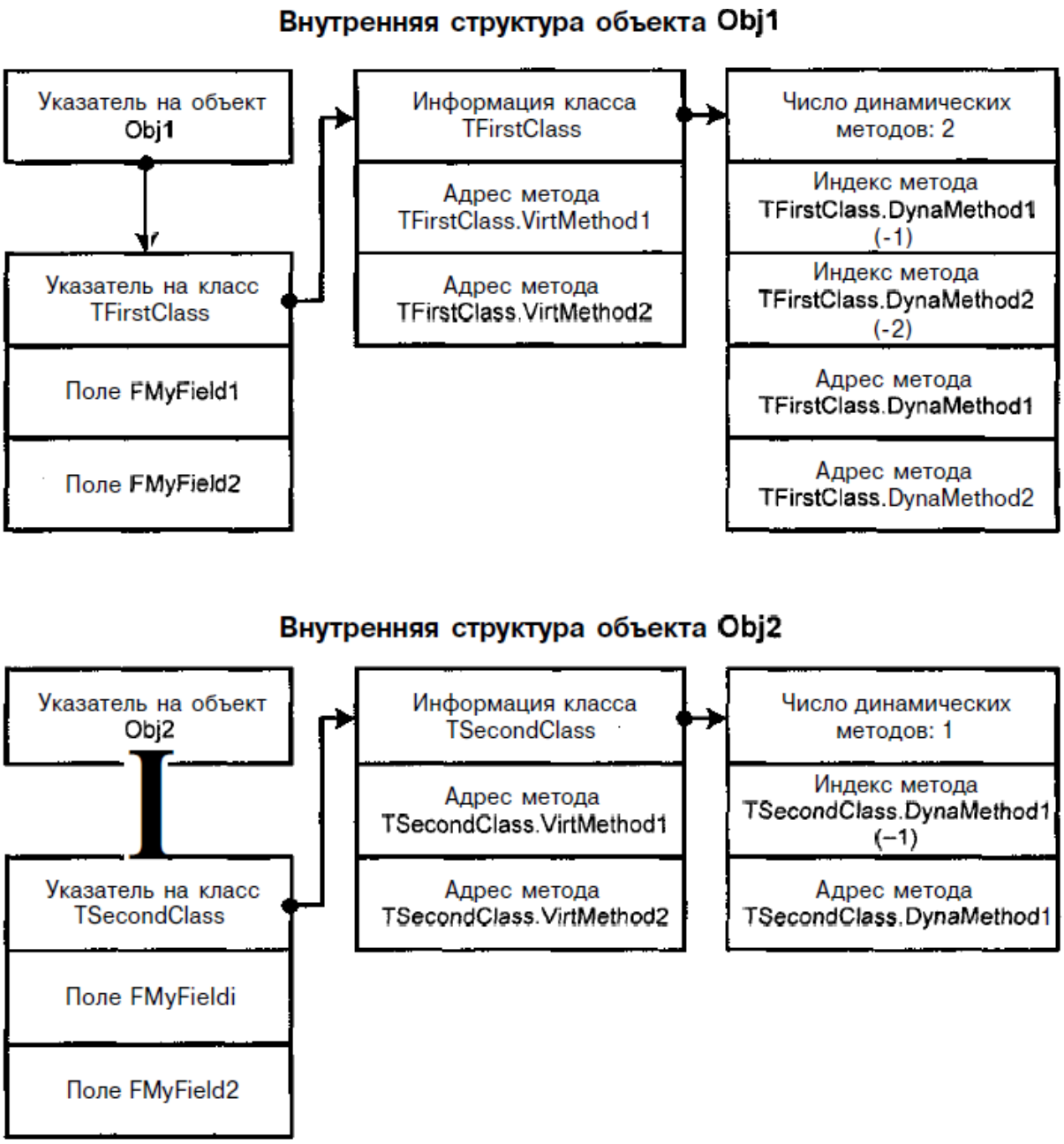

Рис. 1.1. Внутренняя структура объектов Obj1 и Obj2

Выражение вида:

```
AnObject is TObjectType
```

принимает значение True, только если объект AnObject совместим по присваиванию с классом TObjectType, т. е. является объектом этого класса или одного из классов, порожденных от него. Кстати, определенная проверка происходит еще при компиляции: если формально объект и класс несовместимы, то компилятор выдаст ошибку в этом операторе.

Оператор as введен в язык специально для приведения объектных типов. С его помощью можно рассматривать экземпляр объекта как принадлежащий к другому совместимому типу:

```
with ASomeObject as TAnotherType do...
```

От стандартного способа приведения типов с помощью конструкции TAnotherType (ASomeObject) использование оператора as отличается наличием проверки на совместимость типов во время выполнения (как в операторе is): попытка приведения к несовместимому типу приводит к возникновению исключительной ситуации EInvalidCast. После применения оператора as сам объект остается неизменным, но вызываются те его методы, которые соответствуют присваиваемому классу.

Очень полезным может быть оператор as в методах-обработчиках событий.

Для обеспечения совместимости в 99% случаев источник события sender имеет тип TObject, хотя в тех же 99% случаев им является форма или другие компоненты. Поэтому, чтобы иметь возможность пользоваться их свойствами, применяют оператор as:

```
(Sender as TControl).Caption := "Thanks!";
```

Вся информация, описывающая класс, создается и размещается в памяти на этапе компиляции. Возникает резонный вопрос: а нельзя ли получить доступ к ней, не создавая экземпляр объекта? Да, можно. Доступ к информации класса вне методов этого класса можно получить, описав соответствующий указатель, который называется *указателем на класс,* или *указателем на объектный тип* (class reference). Он описывается при помощи зарезервированных слов class of. Например, указатель на класс TObject описан в модуле SYSTEM.PAS и называется TClass:

```
type
   TObject = class;
   TClass = class of TObject;
```

Аналогичные указатели уже описаны и для других важных классов. Вы можете использовать в своей программе TComponentClass, TControlClass и т. п.

Указатели на классы тоже подчиняются правилам приведения объектных типов. Указатель на класс-предок может ссылаться и на любые дочерние классы; обратное невозможно:

```
type
   TFirst = class
   end;
   TSecond = class(TFirst)
   end;
   TFirstClass = class of TFirst;
   TSecondClass = class of TSecond;
var
   AFirst : TFirstClass;
   ASecond : TSecondClass;
begin
   AFirst := TSecond; {допустимо}
   ASecond := TFirst; (недопустимо}
end.
```

С указателем на класс тесно связано понятие *методов класса.* Такие методы можно вызывать без создания экземпляра объекта – с указанием имени класса, в котором они описаны. Перед описанием метода класса нужно поставить зарезервированное слово class:

```
type
   TMyObject= class(TObject)
      class function GetSize: string
   end;

var
   MyObject: TMyObject;
   AString: string;

begin
   AString := TMyObject.GetSize;
   MyObject:= TMyObject.Create;
   AString = MyObject.GetSize;
end.
```

Разумеется, методы класса не могут использовать значения, содержащиеся в полях класса: ведь экземпляра-то не существует. Возникает вопрос: для чего нужны такие методы?

Важнейшие методы класса определены в самом TObject: они как раз и позволяют, не углубляясь во внутреннюю структуру класса, извлечь оттуда практически всю необходимую информацию.