Глава 16. Объектно-ориентированное программирование

Наша природа заключается в том, чтобы действовать, а не быть объектом воздействия.

Стивен Кови.
«Семь навыков высокоэффективных людей»

Первыми появились языки программирования так называемого *императивного* типа, это были хорошо всем знакомые Фортран, Алгол, Бейсик, С, Паскаль и другие. Они реализовывали привычную всем нам методологию (или, как говорят, парадигму) решения задачи: шаги алгоритма обрабатывают данные.[1] Действительно, главным в программе является оператор, он «знает», где находятся подлежащие обработки данные, а сами данные выполняют пассивную роль, они лежат где-то в памяти и «ждут», когда операторы возьмут их из памяти и начнут выполнять над ними операции. Таким образом, как говорят, поток операторов обрабатывает поток данных, по «иностранному» этот принцип работы называется SISD (Single Instruction Single Data).

Внимательные читатели, однако, могли заметить, что наш самый первый исполнитель алгоритма, машина Тьюринга, работала совсем не так. В этой машине обрабатываемые данные (символы, прочитанные с ленты) сами указывали те команды, которые должны их обрабатывать ⚠. Действительно, считанный головкой символ слова сам указывал на клетку таблицы, где располагались три команды, обрабатывающие данный символ. Можно сказать, что поток данных обрабатывался потоком команд, т.е. SDSI (Single Data Single Instruction).

Так устроенные алгоритмические системы называются исполнителями, управляемые потоком данных, а соответствующие компьютеры *потоковыми* ЭВМ (Data Flow Computer). Машина Тьюринга и Нормальные алгоритмы Маркова относятся именно к этому классу алгоритмических систем. Реализовать такие машины «в железе», однако, чрезвычайно трудно по причинам, которые мы здесь не можем рассматривать. Да и языки программирования для таких машин выглядят весьма специфично, больше напоминая нарисованные на плоскости так называемые блок-схемы, которые раньше часто использовались для первоначального (до записи на языке программирования) представления алгоритма.

Тогда и возникло объектно-ориентированное программирование (OOP – object-oriented programming) и объектно-ориентированные языки программирования. Их главная идея была, если не обеспечить главенство данных над операторами, то по крайней мере «уравнять их в правах». Например, пусть у нас есть данное «дверь», меняя такие данные, можно открывать и закрывать двери. В обычном языке программист пишет процедуру «открыть дверь». При вызове этой процедуры ей передаётся параметр (конкретная дверь), и после возврата дверь открыта (а вдруг она уже была открыта 🙄 ?)

В объектно-ориентированном программировании всё наоборот. Сначала программист описывает тип (называемый *классом*) «дверь», к которой приложено действие (называемое *методом*) «открыть». Затем описываются переменные типа «дверь» (называемые *объектами*, реже *экземплярами* класса), к каждому из которых можно послать сообщение «открыть». Например, на языке Free Pascal это выглядит так:

```
type
  Door=object { class }
    procedure Open; { метод }
    begin ... end
  end;
var x,y: Door; { объекты }
begin x.Open; { послать сообщение объекту x }
```

Как видим, посылка сообщения является просто обращением к нужному полю-методу конкретного объекта-двери.

Мы описали класс, используя ключевое слово **object**. Экземпляры (объекты) так описанного класса в языке Free Pascal могут быть любого класса памяти (статического, автоматического или ди-

[1] Слово «императив» (imperativus) чаще всего понимается как «приказ» или «повеление». Таким образом. шаги алгоритма (операторы языка) «приказывают» исполнителю алгоритма выполнять операции над данными.

намического). В частности, такие объекты статического класса автоматически инициализируются нулями, это сами объекты. А вот объекты класса с ключевым словом **class** могут быть только динамического класса, т.е. это ссылочные переменные (ссылки на объекты), хотя для доступа к полям и методам такого объекта и не используется разыменование ↑. Сами объекты типа **class** надо всегда создавать (и инициализировать), вызывая метод `Create` (это аналог процедуры `new`). Аналогично уничтожаются такие объекты методом `Destroy` или методом `Free` (аналог `dispose`). Так, в нашем примере `x` и `y` это объекты, уже расположенные в памяти, их не надо порождать. Оба способы работы с объектами имеют свои достоинства и недостатки. Сейчас, однако, в основном классы описываются с ключевым словом **class**. Ну, как всё запутано 😕.

Сначала в уже существующие языки программирования стали добавляться новые, объектно-ориентированные возможности, так возникли языки C++, Object Pascal и т.д. В новых языках (Java, Python и т.д.) средства объектно-ориентрованного программирования встраивались изначально. Возникли и «чистые» объектно-ориентированные языки (одним из первых был язык Eiffel). В таких языках «обычных» переменных нет совсем, только объекты. Для программистов это может выглядеть немного дико. Например, сложение чисел `x+y` трактуется как: «Послать сообщение методу `summa` класса целых чисел `integer`, этот метод для объектов `x` и `y` порождает новый (безымянный) объект со значением, равным сумме значений объектов `x` и `y`» 🤯.

В качестве примера рассмотрим использование объектно-ориентированного программирования в языке Free Pascal. Все возможности этого режима включаются директивой {$**mode** OBJFPC}. Сначала опишем класс «треугольник», переменные, описанные внутри класса, называются его *членами* (member-fields), или атрибутами, между ними есть тонкое различие, здесь оно не рассматривается, а подпрограммы, как уже говорилось, *методами* (member-functions).

Классы можно рассматривать как обобщение записей, внутри которых, вместе с полями-переменными, можно описывать также константы, свои (внутренние) типы и подпрограммы. Впрочем, внутри подпрограмм нельзя описывать новый тип класса, но можно описать свойство-объект другого класса.

```
{$mode OBJFPC}
type
  Triangle=object { класс }
    const
      Diag='Невозможный треугольник';
{ свойство, TriangleError общее для всех объектов }
      TriangleError: boolean=false;
{ можно описать это свойство и так:
    var TriangleError: boolean; static; }
{ свойства a, b, c – длины сторон }
    var a,b,c: real; { var можно не писать }
{ далее методы }
    constructor init(x,y,z: real);
    function  Square: real;
    procedure SquareOut;
    procedure SidesIn;
    procedure TriangleAdd(x: Triangle);
  end;
constructor Triangle.init(x,y,z: real);
begin
  if (a>=0) and (b>=0) and (c>=0) and
     (a+b>=c) and (a+c>=b) and (b+c>=a)
    then begin a:=x; b:=y; c:=z end
    else begin a:=0; b:=0; c:=0;
      TriangleError:=true; Writeln(Diag)
    end
end;
```

```
function Triangle.Square: real;
  var p: real;
begin { формула Герона }
  p:=(a+b+c)/2;
  Square:=sqrt(p*(p-a)*(p-b)*(p-c))
end;

procedure Triangle.SquareOut;
begin Writeln('Площадь=',Square:8:2) end;

procedure Triangle.SidesIn;
begin
  Write('Введите стороны треугольника:');
{$I-} readln(a,b,c); {$I+}
  if (IOResult<>0)
    then TriangleError:=true else
    if (a<0) or (b<0) or (c<0) or
       (a+b<c) or (a+c<b) or (b+c<a)
      then begin
        writeln(Diag); TriangleError:=true
      end
end;

procedure Triangle.TriangleAdd(x: Triangle);
begin a:=a+x.a; b:=b+x.b; c:=c+x.c end;

var x,y: Triangle;
begin
  x.SidesIn; x.SquareOut; y:=x;
  x.SidesAdd(y); x.SquareOut;
end.
```

Как видим, методы внутри объекта по существу являются объявлениями подпрограмм (но без слова **forvard**), а сами эти подпрограммы описываются ниже, а впереди через точку добавляется имя класса, в который входит данный метод.

Отметим, что, как уже говорилось, описание переменных

```
var x,y: Triangle;
```

для класса, описанного с ключевым словом **object**, порождает в памяти два объекта (а не две ссылки на объекты!). При порождении объектов их свойствам присваиваются значения по умолчанию (для статической памяти нули). Конструктор класса **constructor** позволяет присваивать свойствам объектов заданные начальные значения. В нашем примере после порождении объектов и вызова конструктора

```
var x,y: Triangle;
begin x.init(1,1,1);
```

сторонам треугольника x присваиваются значения (1,1,1), а стороны треугольника y останутся равными (0,0,0).

У каждого объекта свой набор свойств, но некоторые свойства можно сделать общими (существующими в одном экземпляре) для всех объектов класса, для этого надо описать это свойство как переменную с ключевым словом **static**:

```
var d: integer; static;
```

Такого же эффекта можно достичь, если описать свойство в виде типизированной константы, например:

```
const d: integer=1;
```

Класс «заботится» о целостности своих объектов, в нашем случае стороны (a,b,c) должны задавать «настоящий», хотя, возможно, и вырожденный треугольник (например, со сторонами `a=0`,

`b=0` и `c=0`). Это достигается контролем инициализации и ввода сторон треугольников. При возникновении ошибки объект будет вырожденным треугольником (0,0,0), при этом статической переменной класса `TriangleError` присваивается значение **`true`** и выдаётся диагностика `'Невозможный треугольник'`.

Метод `SidesAdd` немного необычен, он «прибавляет» к одному треугольнику другой, под этим понимается сложение соответствующих сторон треугольников.

Докажите, что после такого сложение тоже получится треугольник (контроль не нужен).

Как можно заметить, объекты являются обобщением записей, и можно отметить «творческое» использование оператора присоединения, вместо

```
x.SidesIn; x.SquareOut; y:=x;
x.SidesAdd(y); x.SquareOut;
```

можно писать

```
with x do begin
  SidesIn; SquareOut; y:=x;
  SidesAdd(y); SquareOut
end;
```

Как Вы могли заметить, внутри описания методов такое присоединение делается автоматически.

Важным свойством класса является инкапсуляция (сокрытие) имён внутри описания класса. Как мы уже знаем, имена полей внутри записи «просто так» не видны, нужно использовать запись и селектор нужного поля, например `Writeln(x.a)` для печати длины стороны треугольника. В классе, однако, возможна и более полная инкапсуляция имён, для этого надо подлежащие сокрытию имена описать с модификатором области видимости **`private`**, а остальные имена с модификатором **`public`** (он используется по умолчанию):[1]

```
type
  Triangle=object { класс }
  public
    const
      TriangleError: boolean=false;
    constructor init(x,y,z: real);
    procedure SquareOut;
    procedure TriangleIn;
    procedure TriangleAdd(x: Triangle);
  private
    const Diag='Невозможный треугольник';
{ свойства a, b, c – длины сторон }
    var a,b,c: real;
    function  Square: real;
  end;
```

Модификаторы **`public`** и **`private`** можно указывать и для отдельного свойства или метода, например:

```
var a,b,c: real; public;
function Square: real; private;
```

Внутри класса имена с модификатором **`private`** будут доступны только в том модуле, в котором описан этот класс. Поэтому, достаточно оформить класс и его методы в отдельном модуле (скажем, с именем `Triangle.ppu`) и подключать его с помощью директивы **`uses`** `Triangle;` В этом случае в других модулях такие имена станут полностью недоступными. Например:

```
var x: Triangle;
x.a:=1.0 { ОШИБКА! имя a невидимо! }
```

Можно ещё более ограничить доступ к именам класса, использовав модификатор **`strict private`**, тогда доступ к ним имеют только методы самого этого класса, а не всего модуля.

[1] На самом деле модификаторов больше, остальные мы здесь не рассматриваем.

Другим важным понятием в объектно-ориентированном программировании является наследование. Один класс (потомок) может наследовать свойства и методы другого класса (предка), при необходимости добавляя новые свойства, а также добавляя новые методы и переопределяя старые. Это очень похоже на то, как один блок вкладывается в другой. Внутренний блок получает возможность использовать все имена, описанные во внешнем блоке, заменять часть из них на свои внутренние (локальные) описания, а также добавлять свои собственные. В то же время, наследование имеет более общую природу, например, потомок может непосредственно наследовать сразу от двух предков, что для блоков невозможно. К сожалению, хотя константы, типы и переменные наследуются, но переопределить их в языке Free Pascal невозможно. Также можно запретить наследование всего конкретного класса, поставив при его описании классификатор **sealed**:

```
type Secret=object sealed
     end; { не может иметь наследника }
```

В качестве примера опишем класс «прямоугольный треугольник» (RightTriangle), который будет наследником класса «треугольник» (Triangle):

```
type
  RightTriangle=object(Triangle) { наследник }
  private
    const
      Eps=1e-12;
      Diag1='Плохие катеты и гипотенуза');
    var hc: Real; { высота на гипотенузу }
    function Square: real;
  public
    constructor init(x,y: real);
    procedure SidesIn;
    procedure SidesAdd(x: RightTriangle);
  end;

constructor RightTriangle.init(x,y: real);
begin
  if (a>=0) and (b>=0) then begin
    a:=x; b:=y; c:=sqrt(x*x+y*y); hc:=a*b/c
  end else begin a:=0; b:=0; c:=0;
      TriangleError:=true; Writeln(Diag1)
    end
end;

function RightTriangle.Square: real;
begin Square:=a*b/2 end;

procedure RightTriangle.SidesIn;
begin
  Write('Введите катеты и гипотенузу=');
{$I-} readln(a,b,c); {$I+}
  if (IOResult<>0)
    then TriangleError:=true
  else
    if (a<0) or (b<0) or
       (abs(sqrt(a*a+b*b)-c*c))>Eps) then
    begin
      Writeln(Diag1); TriangleError:=true
    end else { всё хорошо, высота на гипотенузу }
      hc:=a*b/c
end;
```

```
procedure RightTriangle.SidesAdd(x: RightTriangle);
begin
  a:=a+x.a; b:=b+x.b; c:=sqrt(a*a+b*b); hc:=a*b/c
end;
```

Новый класс добавил константы `Eps` и `Diag1`, а также свойство `hc` (высота, опущенная на гипотенузу). Далее, переопределен конструктор (надо проконтролировать размер гипотенузы и присвоить высоту `hc`). Переопределены функция `Square` (для прямоугольных треугольников там всё проще) и процедуры `SidesIn` (другое приглашение на ввод и проверка правильности и диагностика) и `SidesAdd` (складываются длины катетов, а гипотенуза вычисляется заново). В процедуре `SidesIn` добавлено вычисление высоты, опущенной на гипотенузу.

Отметим, что переменные класса-наследника совместимы по присваиванию с переменными класса-предка, обратное неверно:

```
var x: Triangle; y: RightTriangle;
begin
  x.init(1,1,1); y.init(1,2); { y=(1,2,sqrt(5)) }
  x.SidesAdd(x); { x=(2,2,2) }
  y.SidesAdd(y); { y=(2,4,sqrt(20)) }
  { y:=x  – ОШИБКА! } x:=y; { Всё хорошо }
{ в x – объект RightTriangle, x=(2,4,sqrt(20)) }
  y.init(1,1); x.SidesAdd(y);
{ x=(3,5,sqrt(34)),  а вовсе не x=(3,5,sqrt(20)+sqrt(2)) }
```

В «обычных» языках переменная получает тип во время своего порождения и не меняет его вплоть до своего уничтожения. А вот в объектно-ориентированной программировании переменная может менять свой тип в процессе выполнения, это разновидность *полиморфизма*. В нашем примере при порождении переменная `x` имела тип `Triangle`, а затем поменяла его на тип `RightTriangle`. Таким образом, тип переменной зависит от контекста, в этом случае говорят, что тип связывается с переменной не статически, а динамически.[1]

Вам важно понять идеологию объектно-ориентированного программирования. Класс описывает все свойства объекта и задаёт операции по работе с объектом. Над объектами класса разрешается делать только те операции, которые выполняются методами этого класса. Обратите внимание, что программист, использующий объект «треугольник», не только не может читать и изменять его стороны, он даже не знает, как они называются! Действительно, если дать возможность «обычному» программисту менять длину стороны треугольника, то он может перестать быть треугольником, т.е. объект «испортиться». Когда же классу по каким-то причинам надо дать возможность для пользователя читать, скажем, сторону `a`, то он предоставляет для этого метод:

```
public function Side_a: real;
begin Side_a:=a end;
```

Как следствие ограничения видимости, класс может обеспечить целостность своих объектов. Треугольник порождается вырожденным, с «хорошими» сторонами (их начальные значения 0, 0, 0). Прямоугольный треугольник порождается и с правильной высотой на гипотенузу. Далее все изменения треугольника жёстко контролируются методами класса, никто нам его не испортит ⚠.

Это самое элементарное введение в объектно-ориентированное программирование, на самом деле там всё **много сложнее**, и Вам это ещё предстоит изучать.

[1] Как ранее упоминалось, динамическую типизацию имеет тип `Variant` языка Free Pascal, и все переменные языка Python, они могут менять свой тип после каждого присваивания им нового значения.