Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А. А. Волосевич

ООП НА ЯЗЫКЕ OBJECT PASCAL

Курс лекций
для студентов специальности
1-40 01 03 Информатика и технологии программирования

Минск 2013

Содержание

1. Базовые понятия ООП

Парадигма программирования – это система идей и понятий, определяющих стиль создания компьютерных программ. Примерами парадигм являются императивное программирование, структурное программирование, объектно-ориентированное программирование. В *императивном программировании* процесс вычисления описывается в виде последовательности команд, которые должен выполнить компьютер. *Структурное программирование* основано на представлении программы в виде иерархической структуры связанных блоков (подзадач). Усложнение программного обеспечения привело к широкому распространению *объектно-ориентированного программирования* (ООП). Эта парадигма предлагает рассматривать программу как процесс взаимодействия некоторых вполне самостоятельных единиц, по аналогии с реальным миром называемых объектами. Создание программы заключается в наиболее полном описании соответствующих объектов и кодировании связей между ними. ООП оказалось настолько продуктивной идеей, что большинство современных языков программирования являются либо чисто объектно-ориентированными (Java, C#), либо содержат средства ООП в качестве надстройки (C++, Object Pascal).

ООП основывается на следующих принципах: *абстракция*, *инкапсуляция*, *наследование*, *полиморфизм*. Абстракция в ООП – это набор наиболее значимых характеристик объекта (способ выделения подобных характеристик называется *абстрагированием*). Инкапсуляция – это логическое объединение в одном программном типе, называемом *класс*, как данных, так и подпрограмм для их обработки. Данные класса хранятся в *полях класса*, подпрограммы для работы с полями называются *методами класса*.

Разберём синтаксис определения класса в Object Pascal. В качестве примера рассмотрим класс для описания человека (нас интересуют имя и возраст):

```
type TPerson = class
       fName: string;
       fAge: Integer;
       procedure SetAge(Age: Integer);
       function SayName: string;
     end;
```

Определение класса размещается в секции описания типов. Это может быть глобальная секция типов программы, секция типов в интерфейсной части модуля или в разделе реализации, но не секция типов в подпрограмме. Для определения класса используется ключевое слово `class`. Вначале описываются поля класса. Полями класса `TPerson` являются `fName` и `fAge`[1]. После описания всех полей следует указание методов класса. Методы `TPerson` – это процедура `SetAge` и функция `SayName`. Класс содержит только заголовки методов, реализация методов описывается отдельно. Если класс описан в модуле, реализация методов должна нахо-

[1] Для имён полей в Object Pascal традиционно используется префикс «f» (field – поле).

диться в том же модуле в секции `implementation`. Чтобы показать, что подпрограмма является реализацией метода класса, используется синтаксис *`имя-класса.имя-метода`*. В теле методов обращение к полям класса происходит без указания каких-либо дополнительных спецификаторов:

```
procedure TPerson.SetAge(Age: Integer);
begin
  if Age > 0 then fAge := Age
end;

function TPerson.SayName: string;
begin
  Result := 'My name is ' + fName
end;
```

После того как класс описан, можно объявить переменную класса, называемую *экземпляром класса* или *объектом*:

```
var Man: TPerson;
```

В реальной жизни классу соответствует абстрактное понятие, которое уточняется в своих проявлениях. Так, класс `TPerson` как бы пытается описать человека вообще, через его параметры и действия, производимые им и над ним. Объект `Man` – это конкретный человек, описанный определёнными значениями полей.

В Object Pascal для работы с полями и методами объекта используется синтаксис *`имя-объекта.имя-поля-или-метода`*. Это напоминает работу с переменной типа запись:

```
Man.fName := 'John Dow';
Man.SetAge(37);
writeln(Man.SayName, Man.fAge);
```

Приведём полный пример консольного приложения, содержащего определение класса и демонстрирующего работу с объектами:

```
program OOPExample;
{$APPTYPE CONSOLE}

type TPerson = class
       fName: string;
       fAge: Integer;
       procedure SetAge(Age: Integer);
       function SayName: string;
     end;

procedure TPerson.SetAge(Age: Integer);
begin
  if Age > 0 then fAge := Age
end;
```

```
function TPerson.SayName: string;
begin
  Result := 'My name is ' + fName
end;

var Man: TPerson;

begin
  Man := TPerson.Create;     // создание объекта
  Man.fName := 'John Dow';
  Man.SetAge(37);
  writeln(Man.SayName, Man.fAge);
end.
```

2. Создание и уничтожение объектов

Object Pascal использует так называемую *ссылочную объектную модель*. Это означает, что все объекты размещаются в памяти динамически, а объектные переменные фактически являются указателями на данные объекта в динамической памяти и имеют одинаковый размер (4 байта). Однако для доступа к данным объекта не используется символ разыменовывания ^ (записывается `Man.fName`, а не `Man^.fName`, хотя подразумевается именно второе). Для начального размещения объектов в динамической памяти служит особый вид методов, называемых *конструкторами*.

Чтобы объявить метод как конструктор, используется ключевое слово `constructor`. Оно записывается вместо слова `procedure` при определении метода в классе и при реализации метода (конструктор не может быть функцией). Компилятор автоматически добавляет к телу конструктора код, выделяющий в динамической памяти участок для полей объекта и обнуляющий этот участок. Так как конструктор необходимо выполнить перед использованием объекта, то в тело конструктора обычно помещают операторы инициализации объекта, например, задание начальных значений для полей. Отметим, что Object Pascal допускает существование в классе нескольких конструкторов. Традиционное (но не обязательное) имя для конструктора – `Create`.

Добавим конструктор в класс `TPerson`:

```
type TPerson = class
       . . .
       constructor Create;
     end;

constructor TPerson.Create;
begin
  fAge := 1;
  fName := 'Person'
end;
```

Синтаксис вызова конструктора: *имя-объекта* := *имя-класса.имя-конструктора*:

```
Man := TPerson.Create;
```

Конструктор можно вызывать в виде *имя-объекта.имя-конструктора*, т. е. как обычный метод:

```
Man.Create;
```

Такой вызов означает простое выполнение тела конструктора (ре-инициализацию полей). Его можно применять только для тех объектов, которые уже размещены в памяти.

Если объект не используется, то занимаемая им динамическая память должна быть освобождена. Для уничтожения объектов предназначены особые методы – *деструкторы*.

Для объявления деструкторов используется ключевое слово `destructor`. Тело деструктора – подходящее место для финальных действий с объектом. Традиционное имя для деструктора – `Destroy`.

Добавим в класс `TPerson` деструктор:

```
type TPerson = class
       . . .
       constructor Create;
       destructor Destroy;
     end;

destructor TPerson.Destroy;
begin
  fAge := 0;
  fName := ''
end;
```

Теперь полный цикл работы с объектом `Man` выглядит следующим образом:

```
Man := TPerson.Create;    // создание объекта
Man.fName := 'John Dow';  // работа с объектом
Man.Destroy;              // уничтожение объекта
```

Деструктор можно вызвать только у инициализированного объекта. Попытка вызвать деструктор у неинициализированного объекта может привести к исключительной ситуации в работе программы.

3. Параметр self

Вернёмся к рассмотрению класса `TPerson`. Этот класс содержит два поля и два метода. Представим, что имеется 100 объектов класса `TPerson`. Так как каждый объект конкретизируется значениями своих полей, то в памяти должно содержаться 100 наборов полей класса. Означает ли это, что и код методов класса

будет продублирован 100 раз? Определённо, нет. Код методов класса содержится в памяти в единственном числе, как и код любой подпрограммы. Однако как метод определяет, с полями какого объекта он работает?

```
var A, B: TPerson;
. . .
A.SetAge(10);  // SetAge работает с A.fAge
B.SetAge(40);  // SetAge работает с B.fAge
```

Для выявления конкретного объекта, с которым происходит работа, любому методу передаётся *скрытый параметр* `self`. Этот параметр указывает на объект, вызывающий метод. Тип параметра `self` совпадает с типом класса. Например, на уровне компилятора описание метода `SetAge` и работу с ним можно представить следующим образом:

```
procedure TPerson.SetAge(Age: Integer; self: TPerson);
begin
  if Age > 0 then self.fAge := Age
end;
. . .
TPerson.SetAge(10, A);
TPerson.SetAge(40, B);
```

Подчеркнём два важных момента:

1. Параметр `self` передаётся в любой метод;
2. Методы можно воспринимать как обыкновенные подпрограммы, которые принимают дополнительный параметр `self`.

Практически всегда в явном употреблении `self` нет необходимости. Одно из исключений – использование одинаковых идентификаторов для полей класса и параметров метода. Предположим, что класс `TPerson` содержит поле с идентификатором `Age`, а не `fAge`. Тогда корректная реализация метода `TPerson.SetAge` должна выглядеть так:

```
procedure TPerson.SetAge(Age: Integer);
begin
  // Age – параметр метода, self.Age – поле объекта
  if Age > 0 then self.Age := Age
end;
```

4. Ограничение видимости элементов класса

Класс `TPerson` содержит четыре *элемента класса* – два поля и два метода. И поля, и методы без затруднений доступны из любого объекта:

```
var Man: TPerson;
. . .
Man.SetAge(37);    // работа с полем через метод
Man.fAge := -100;  // непосредственная работа с полем
```

Приведённый пример показывает недостаток подобной свободы. Одно из назначений метода `TPerson.SetAge` – корректная установка возраста. Вместе с тем, этот метод можно обойти, установив возраст прямо через поле.

В Object Pascal существуют специальные *директивы ограничения видимости*, которые при описании класса позволяют контролировать видимость его элементов. Эти директивы разделяют объявление класса на *секции видимости*.

Рассмотрим две директивы ограничения видимости – `private` и `public`. Все элементы класса из секции `private` доступны для использования только в том модуле или программе, которые содержат объявление класса. В секции `private` обычно размещаются поля и методы, описывающие внутренние особенности реализации класса. Элементы из секции `public` не имеют ограничений на использование.

Подчеркнём следующие особенности использования директив ограничения видимости. Во-первых, внутри модуля с описанием класса директивы не действуют. Во-вторых, по умолчанию для элементов класса установлена директива видимости `public`. И, наконец, порядок директив в описании класса произволен, допускается повторение директив. Однако принято описывать элементы класса в порядке увеличения открытости.

Применим директивы `private` и `public` к классу `TPerson`, поместив его в отдельный модуль. Обратите внимание: описание класса помещено в секцию `interface`, а реализация – в секцию `implementation`:

```
unit TPersonClass;

interface

type TPerson = class
     private
       fName: string;
       fAge: Integer;
     public
       procedure SetAge(Age: Integer);
       function SayName: string;
     end;

implementation

procedure TPerson.SetAge(Age: Integer);
begin
  if Age > 0 then fAge := Age
end;

function TPerson.SayName: string;
begin
  Result := 'My name is ' + fName
end;
end.
```

Сейчас попытка обратится к полю `fAge` в программе, использующей класс `TPerson`, вызовет ошибку компиляции:

```
program OOPExample;
uses TPersonClass

var Man: TPerson;
. . .
Man.fAge := -100;  // ошибка компиляции!
```

Таким образом, при помощи директив поля класса защищены от «вторжения» пользователей. Однако теперь у `TPerson` появился недостаток – значения полей нельзя прочитать непосредственно. Этот недостаток устраняется введением в класс дополнительных методов. Это типично для ООП – осуществлять доступ к полям только через методы класса. Говорят, что такие методы составляют *интерфейс класса*.

```
type TPerson = class
     private
       fName: string;
       fAge: Integer;
     public
       procedure SetAge(Age: Integer);
       function GetAge: Integer;
       procedure SetName(Name: string);
       function SayName: string;
     end;
```

Обсудим преимущества, которые даёт использование методов для доступа к полям. Первое преимущество: при использовании методов данные объекта могут быть представлены в разных форматах без дублирования. Для иллюстрации рассмотрим простой класс `TTemperature`, назначение которого – хранить данные о температуре. В классе будет использоваться две пары методов для чтения и записи поля, что позволит получать температуру в двух вариантах – градусах Цельсия и кельвинах:

```
type TTemperature = class
     private
       fTemp: double;
     public
       procedure SetTempCelsius(Temp: double);
       function GetTempCelsius: double;
       procedure SetTempKelvin(Temp: double);
       function GetTempKelvin: double;
     end;
. . .
```

```
procedure TTemperature.SetTempCelsius;
begin
  fTemp := Temp + 273.15
end;

function TTemperature.GetTempCelsius;
begin
  Result := fTemp - 273.15
end;

procedure TTemperature.SetTempKelvin;
begin
  fTemp := Temp
end;

function TTemperature.GetTempKelvin;
begin
  Result := fTemp
end;
. . .
var T: TTemperature;
. . .
T.SetTempCelsius(20);  // установили в градусах Цельсия
A := T.GetTempKelvin;  // прочитали в кельвинах
```

Второе преимущество использования методов для доступа к полям класса: разработчик класса может незаметно для пользователей изменять структуру хранения данных класса. Например, в классе `TTemperature` мы могли бы хранить температуру в градусах Цельсия. Для этого нам пришлось бы переписать методы класса. Но если оставить их заголовки прежними, пользователь класса подобной замены не увидит.

Подводя итог, можно сформулировать следующее правило: *создатель класса должен исключить возможность прямой работы с полями – для этого следует использовать набор интерфейсных методов*.

5. Свойства

В предыдущем параграфе обсуждались преимущества, которые дают интерфейсные методы класса. Однако с точки зрения пользователя класса применение методов для доступа к полям имеет небольшой недостаток – громоздкость вызова. Развитие концепций ООП привело к появлению понятия *свойства* (property). Свойство – это элемент класса, работа с которым происходит так же, как с полем объекта. Разница между полем и свойством заключается в следующем: обращение к свойству компилятор транслирует в обращение к полю или в вызов метода, следовательно, при работе со свойствами могут выполняться некоторые действия.

В Object Pascal для объявления свойства используется ключевое слово `property`. Далее следует имя свойства и указывается его тип. После директивы `read`

указывается имя поля или метода для чтения свойства. После директивы `write` указывается имя поля или метода для записи свойства. Считается, что свойство записывается, когда ему присваивается некое значение, иначе свойство читается. Тип полей, употребляемых после `read` и `write`, должен совпадать с типом свойства. Метод, используемый для чтения простого свойства, должен быть функцией без параметров, тип возвращаемого значения которой совпадает с типом свойства. Метод для записи – процедура с одним параметром, имеющим тип свойства. Принято соглашение, согласно которому имена методов чтения свойств начинаются с префикса `Get`, а имена методов записи – с префикса `Set`. Объявление свойства может следовать только после объявления полей и методов, которые используются свойством.

Добавим свойства в класс `TPerson`:

```
type TPerson = class
     private
       fName: string;
       fAge: Integer;
     public
       procedure SetAge(Age: Integer);
       function GetAge: Integer;
       procedure SetName(Name: string);
       function SayName: string;
       property Age: Integer read GetAge write SetAge;
       property Name: string read SayName write SetName;
     end;
```

Работу со свойствами демонстрирует следующий фрагмент кода:

```
var Man: TPerson;
. . .
Man.Name := 'Alex';  // транслируется в Man.SetName('Alex')
Man.Age := 101;      // транслируется в Man.SetAge(101)
```

Подчеркнём, что свойства класса призваны облегчить работу с объектом для пользователя. Фактически, свойства «живут» только до компиляции программы, во время которой заменяются методами или полями. В отличие от полей свойства не занимают места в памяти. Это накладывает определённые ограничения на их использование. Свойства нельзя передавать в качестве `var`-параметров в подпрограммы, к ним нельзя применять операцию взятия адреса.

Употребление свойств позволяет «сэкономить» на методах класса. Например, в классе `TPerson` методы `GetAge` и `SetName` введены только для того, чтобы обеспечить полный доступ к полям, никаких особых действий они не выполняют. Заменим в определении свойств эти методы полями `fAge` и `fName`. В свою очередь, методы, обслуживающие свойства, поместим в секцию `private`, так как именно свойства теперь будут составлять интерфейс класса:

```
type TPerson = class
     private
       fName: string;
       fAge: Integer;
       procedure SetAge(Age: Integer);
       function SayName: string;
     public
       property Age: Integer read fAge write SetAge;
       property Name: string read SayName write fName;
     end;
```

Разберём некоторые нюансы описания и использования свойств. Если опустить директиву `read`, можно получить свойство только для записи. Если опустить `write`, получим свойство, значение которого можно читать, но не записывать. Однако какая-то из директив должна в объявлении свойства присутствовать.

Рассмотрим пример класса `TArray`. Он будет представлять массив вещественных чисел, в котором кроме самих чисел хранится количество элементов, а также имеется свойство для получения максимального элемента. Начальный вариант такого класса может выглядеть так:

```
type TArray = class
     private
       fData: array[1..1000] of double;  // массив «с запасом»
       fLength: Integer;  // реальная длина массива
     public
       procedure WriteElement(Ind: Integer; Value: double);
       function ReadElement(Ind: Integer): double;
       function FindMax: double;
       property Length: Integer read fLength;
       property Max: double read FindMax;
     end;

procedure TArray.WriteElement(Ind: Integer; Value: double);
begin
  if (Ind > 0) and (Ind <= 1000) then
  begin
    fData[Ind] := Value;
    if Ind > fLength then fLength := Ind;
  end;
end;

function TArray.ReadElement(Ind: Integer): double;
begin
  if (Ind > 0) and (Ind <= fLength)
  then Result := fData[Ind]
  else Result := 0
end;
```

```
function TArray.FindMax: double;
var i: Integer;
begin
  Result := fData[1];
  for i := 2 to fLength do
    if fData[i] > Result then Result := fData[i]
end;
. . .
var Mas: TArray;
. . .
Mas.WriteElement(1, 10);
Mas.WriteElement(2, 100);
Mas.WriteElement(10, 1);
k := Mas.Max;     // k = 100
l := Mas.Length;  // l = 10
```

Обратите внимание: свойства `Length` и `Max` являются свойствами только для чтения. Более того, свойство `Max` вообще можно считать «виртуальным», так как оно не связано ни с одним полем класса (но пользователь класса этого не заметит).

Для пользователя класса `TArray` естественным желанием является получить более удобный способ доступа к данным в `fData`. Object Pascal разрешает объявлять *свойства-массивы*. Подобные свойства используются в основном в классах, данные которых подразумевают доступ с использованием различных индексаторов. Добавим свойство-массив в класс `TArray` (и перенесём методы `WriteElement` и `ReadElement` в секцию `private`):

```
type TArray = class
     private
     . . .
       procedure WriteElement(Ind: Integer; Value: double);
       function ReadElement(Ind: Integer): double;
     public
     . . .
       property Data[I: Integer]: double read ReadElement
                                          write WriteElement;
     end;
```

Для объявления свойства-массива после имени свойства в квадратных скобках указывается имя и тип индекса. Как и для индексированных свойств, для доступа к свойствам-массивам возможно только использование методов. Первый параметр этих методов должен совпадать по типу с индексом свойства. Ниже приведён пример работы со свойством `Data` и указано, во что транслируются обращения к нему:

```
for i := 1 to 5 do
  Mas.Data[i] := 1000;  // транслируется в Mas.WriteElement(i, 1000)
```

Тип индекса свойства-массива не ограничен диапазоном (индекс может быть любого типа – строка, вещественное число, класс). Допускается работа только с элементами свойства-массива, а не со всем свойством целиком.

Свойства-массивы могут быть многомерными. В этом случае количество необходимых параметров у методов чтения и записи увеличивается на соответствующее число.

Если при описании свойства-массива добавить в конце директиву `default`, то такое свойство становится *основным свойством класса*. Для основного свойства можно указывать индекс непосредственно после имени объекта, не используя идентификатор свойства. Сделаем свойство `Data` основным:

```
type TArray = class
     . . .
       property Data[I: Integer]: double read ReadElement
                                  write WriteElement; default;
     end;
```

Теперь с ним можно работать так:

```
for i := 1 to 5 do
  Mas[i] := 1000;  // вместо Mas.Data[i] := 1000
```

Только свойство-массив может быть основным свойством класса. У класса может быть только одно основное свойство.

6. Отношения между классами и объектами

Если предположить, что классы и объекты в ООП являются отражением понятий реальной жизни, то легко можно установить те отношения, которые могут существовать между классами. Первый тип отношений соответствует ситуации, когда один класс включает в себя объекты других классов. Такой тип отношений называется *агрегированием* (иначе называемым *отношением has-a* или *отношением part-of*). Данный тип отношения моделируется включением в класс полей-объектов. Например, класс для представления группы людей может иметь следующее описание:

```
type TCrowd = class
       fPeople: array[1..100] of TPerson;
       . . .
     end;
```

Для классов, реализующих агрегирование, конструктор, как правило, занимается созданием объектов-полей, а деструктор уничтожает эти объекты:

```
constructor TCrowd.Create;
var i: Integer;
begin
  for i := 1 to 100 do
```

```
    fPeople[i] := TPerson.Create
end;
```

Следующий тип отношений связан с ситуацией, когда понятие, соответствующее одному классу, уточняется понятием, соответствующим другому классу. Пусть необходим класс для описания служащих – `TEmployee`. Можно рассуждать так: любой служащий является человеком (`TPerson`), но служащий – это такой человек, который получает зарплату. Отношение между классами `TEmployee` и `TPerson` называется *наследованием* (*отношение is-a*). Наследование является одним из базовых принципов ООП. Наследование предполагает создание новых классов на основе существующих. В нашем случае мы можем не писать класс `TEmployee` «с нуля», а воспользоваться классом `TPerson` как основой. При наследовании новый класс называется *классом-потомком* (или *дочерним классом*, *производным классом*), старый – *классом-предком* (или *родительским классом*, *базовым классом*). При помощи наследования можно строить так называемое *дерево классов* (или *иерархию классов*), последовательно уточняя описание класса и переходя от общих понятий к частным.

Рассмотрим синтаксис наследования классов. В Object Pascal при объявлении класса-потомка имя класса-предка указывается после ключевого слова `class` в круглых скобках. Описание класса-потомка включает только те элементы, которых нет в предке, так как потомок получает все элементы предка автоматически.

Описание класса `TEmployee` может выглядеть следующим образом:

```
type TEmployee = class(TPerson)
     private
       fSalary: double;
       procedure SetSalary(Value: double);
     public
       property Salary: double read fSalary write SetSalary;
     end;
```

Как наследник, `TEmployee` содержит все поля, методы и свойства `TPerson` и, кроме этого, добавляет собственное поле `fSalary`, метод `SetSalary` и свойство `Salary`.

Объекты классов-потомков совместимы по присваиванию с объектами классов-предков. При этом действует следующее правило: *объекту родительского класса можно присвоить объект дочернего класса, но не наоборот*:

```
var Man: TPerson; Employee: TEmployee;
. . .
Man := Employee;  // допустимо
Employee := Man;  // ошибка компиляции
```

Обосновывается вышеуказанное правило следующим образом. Так как дочерний класс может добавлять к родительскому классу новые поля, то при присваивании объекту дочернего класса объекта родительского класса обращение к новым полям приведёт к выходу за границу памяти объекта родительского класса. Это является недопустимым.

Отметим следующие особенности объектной модели Object Pascal. Наследование в Object Pascal разрешено только от одного предка. Все классы в Object Pascal имеют одного общего предка. Таким предком является класс `TObject`. Объявления `TPerson = class` и `TPerson = class(TObject)` полностью эквивалентны. Класс `TObject` содержит пустой конструктор `Create` и пустой деструктор `Destroy`, которые можно применять в простых классах (этот факт использовался в некоторых предыдущих примерах).

С наследованием связана директива ограничения видимости `protected`. Элементы класса из секции `protected` могут использоваться вне пределов модуля с объявлением класса, но только потомками класса.

7. Перекрытие методов

Рассмотрим класс для описания животных, способных издавать звуки:

```
type TPet = class
       procedure Speak;
     end;

procedure TPet.Speak;
begin
  Beep;  // ничего особенного – просто звуковой сигнал
end;
```

Потомки `TPet` – классы `TDog` и `TCat` – тоже способны «издавать звуки», но делают это по-своему. Ситуация является достаточно типичной при проектировании иерархии классов: наследник имеет такие же методы как и предок, но реализует их своим способом. ООП даёт возможность описать в классе-потомке метод с тем же именем, что и в классе-предке, но с собственной реализацией. Это называется *перекрытие* (*замещение*) методов:

```
type TDog = class(TPet)
       procedure Speak;  // перекрытие метода TPet.Speak
     end;

     TCat = class(TPet)
       procedure Speak;  // перекрытие метода TPet.Speak
     end;

procedure TDog.Speak;
begin
  writeln('Bow-wow')
end;
```

```
procedure TCat.Speak;
begin
  writeln('Mew')
end;
```

При перекрытии методов их сигнатуры (то есть количество и тип параметров) могут различаться.

Если дочерний класс не перекрывает метод предка, а добавляет новый метод с тем же именем, необходимо воспользоваться директивой `overload`, чтобы явно указать на перегрузку. Естественно, такие методы должны различаться сигнатурой:

```
type TCatWithVolume = class(TCat)
       // не перекрыли TCat.Speak, а добавили метод с параметром
       procedure Speak(Volume: Integer);overload;
     end;
```

Разберём вопрос о перекрытии полей класса. Классическое ООП не допускает подобного. Однако в Object Pascal дочерний класс может объявить поле с тем же именем, что и поле в родительском классе, но с другим типом. В этом случае методы в дочернем классе будут работать с новым полем, методы в родительском классе – со старым. На практике такая возможность практически не используется, так как способна основательно запутать и проектировщика, и пользователей класса.

Достаточно часто метод класса-потомка не заменяет действия в методе класса-предка, а дополняет их. Чтобы не дублировать код, в методе класса-потомка можно вызвать метод класса-предка. Для вызова перекрытых методов ближайшего класса-предка применяется конструкция `inherited` *`имя-метода-класса-предка`*. Если имя и параметры вызываемого у предка метода совпадают с именем и параметрами вызывающего метода, достаточно записать только ключевое слово `inherited`:

```
procedure TCat.Speak;
begin
  inherited;  // вызов TPet.Speak – звуковой сигнал
  writeln('Mew')
end;
```

Если в непосредственном предке метод, вызываемый через `inherited`, отсутствует, компилятор проводит поиск такого метода по иерархии классов вплоть до корневого класса. Если метод не найден, никаких действий не предпринимается.

В иерархии классов работа конструктора класса-потомка, как правило, начинается с вызова конструктора класса-предка для корректной инициализации полей предка. Добавим конструктор в класс `TEmployee` (используется вариант класса `TPerson` с простым конструктором):

```
type TEmployee = class(TPerson)
     public
       . . .
       constructor Create;
     end;

constructor TEmployee.Create;
begin
  inherited;  // fAge := 1, fName := 'Person'
  fName := 'Employee'
end;
```

Для деструкторов в иерархии классов действует правило, согласно которому вызов унаследованного деструктора происходит в конце работы класса-потомка.

8. Полиморфизм

Полиморфизм является одним из трёх принципов ООП. Прежде чем дать точное определение полиморфизма и рассмотреть синтаксис его реализации в Object Pascal, изучим ситуации, приводящие к этому понятию.

Опишем простую иерархию классов для графических объектов – базовый класс `TFigure` и его наследники `TSquare` и `TCircle`. Наделим `TFigure` методом рисования `Draw` и методом стирания фигуры `Hide`. Естественно, классы `TSquare` и `TCircle` перекрывают эти методы (у класса `TFigure` методы будут пустыми):

```
type TFigure = class
       // схематическое описание класса
       procedure Draw;
       procedure Hide;
     end;

     TSquare = class(TFigure)
       procedure Draw;
       procedure Hide;
     end;

     TCircle = class(TFigure)
       procedure Draw;
       procedure Hide;
     end;

procedure TFigure.Draw;
begin
end;

procedure TFigure.Hide;
begin
end;
```

```
procedure TSquare.Draw;
begin
  // здесь рисуем квадрат
end;

procedure TSquare.Hide;
begin
  // здесь стираем квадрат
end;

procedure TCircle.Draw;
begin
  // рисуем круг
end;

procedure TCircle.Hide;
begin
  // стираем круг
end;
```

Рассмотрим следующую ситуацию. Пусть имеется подпрограмма (или метод некоего класса), в которой происходит рисование графического объекта. Этот объект передаётся подпрограмме в качестве параметра. Каким должен быть тип этого параметра? Следуя правилу присваивания объектов, заключаем, что типом параметра должен быть `TFigure`, что позволит передать в подпрограмму объекты любых его дочерних классов. Заголовок подпрограммы может выглядеть следующим образом:

```
procedure WorkWithObjects(X: TFigure);
begin
  . . .
  X.Draw
end;
```

Использовать подпрограмму `WorkWithObjects` можно так:

```
var A: TCircle;
    B: TSquare;
. . .
A := TCircle.Create;
B := TSquare.Create;
WorkWithObjects(A);
WorkWithObjects(B);
```

Небольшое отступление: обратите внимание на использование конструкторов для создания объектов. Так как конструкторы в `TCircle` и `TSquare` не описывались, то в данном примере используется конструктор `TObject.Create`, доставшийся этим классам «по наследству». Однако этот конструктор создаёт именно

объекты класса TCircle и TSquare. Если бы эти классы имели разный набор полей, для объектов резервировался бы разный объем динамической памяти. Напомним, что выделением памяти для объекта занимается некий дополнительный код, присутствующий в любом конструкторе неявно.

Вернёмся к нашему примеру. Хотя он синтаксически корректен и компилируется, работать он будет неверно. Несмотря на тип фактических параметров процедуры WorkWithObjects, при работе эта подпрограмма будет вызывать метод TFigure.Draw.

Смоделируем вторую ситуацию. Пусть имеется массив из объектов класса TFigure или его наследников. Инициализируем такой массив и попытаемся нарисовать все графические объекты в цикле:

```
var Figures: array[1..3] of TFigure;
. . .
// корректно по правилам присваивания для объектов
Figures[1] := TCircle.Create;
Figures[2] := TSquare.Create;
Figures[3] := TCircle.Create;

// пытаемся нарисовать все объекты в цикле
for i := 1 to 3 do
  Figures[i].Draw;
```

Ожидается, что в результате работы цикла будут нарисованы круг, квадрат и круг. Тем не менее, будет получена лишь последовательность из трёх вызовов TFigure.Draw.

Ещё одна ситуация. Добавим в класс TFigure метод для перемещения фигур. Перемещение фигуры – это стирание фигуры, изменение её координат и рисование фигуры на новом месте:

```
type TFigure = class
       . . .
       procedure Move;
     end;

procedure TFigure.Move;
begin
  Hide;
    // здесь изменили координаты фигуры
  Draw
end;
```

Логика работы метода Move сохраняется для дочерних классов TCircle и TSquare. Значит, переопределять этот метод в дочерних классах не требуется. Однако следующий вызов приведёт не к перемещению квадрата, а к вызову TFigure.Hide, изменению координат (квадрата) и вызову TFigure.Draw:

```
var Square: TSquare;
. . .
Square.Move;  // увидим, что квадрат не переместился!
```

Подобные расхождения между желаемым поведением объектов и действительным возникают из-за того, что в наших примерах использовались *статические методы*[1]. Адрес статического метода вычисляется на этапе компиляции. Он определяется классом объекта и не зависит от того, с каким классом будет связан объект на этапе выполнения. Рассмотрим строку вызова `X.Draw` из процедуры `WorkWithObjects`. Во время компиляции программы компилятор пытается определить тип переменной `X`. Он может сделать это по объявлению переменной в заголовке процедуры. Тип `X` – это `TFigure`, значит запись `X.Draw` означает вызов метода `TFigure.Draw` (если вспомнить о неявном параметре `self`, то более точно – `TFigure.Draw(X)`).

Нам необходимо, чтобы поведение объекта и вызов его методов определялись непосредственно в период выполнения программы. Это означает, что адрес метода должен вычисляться непосредственно в период выполнения по тому типу, который объект имеет в данный момент. Подобным образом работают *виртуальные методы*. Они функционируют по следующей схеме. Каждый объект, наряду со значениями своих полей, хранит указатель на специальную *таблицу виртуальных методов* (virtual method table, VMT). Таблица виртуальных методов индивидуальна и единственна для каждого класса. В ней хранятся адреса всех виртуальных методов класса (как собственных, так и унаследованных). Связь между объектом и VMT класса осуществляется во время начальной инициализации объекта, то есть при вызове конструктора. Виртуальные методы идентифицируются по константе-смещению в VMT. Во время выполнения программы из экземпляра объекта извлекается указатель на VMT и, используя константу-смещение, вычисляется адрес необходимого метода.

Для объявления виртуального метода используется директива `virtual`. Исходный метод, объявленный как виртуальный в классе-предке, перекрывается в классе-потомке методом с тем же именем и параметрами и помечается директивой `override`. Если хотя бы одно из этих требований не соблюдено, связь между виртуальными методами в иерархии классов теряется. При реализации методов директивы `virtual` и `override` не указываются.

Отредактируем классы `TFigure`, `TSquare` и `TCircle`, сделав их методы виртуальным:

```
type TFigure = class
       procedure Draw;virtual;
       procedure Hide;virtual;
     end;
```

[1] Термин специфичен для Object Pascal, в котором статические методы противопоставляются виртуальным. В других языках программирования под статическим методом обычно понимают методы, работающие с классом, а не объектом.

```
TSquare = class(TFigure)
  procedure Draw;override;
  procedure Hide;override;
end;

TCircle = class(TFigure)
  procedure Draw;override;
  procedure Hide;override;
end;
```

Если внести подобные изменения в описания классов, то во всех перечисленных ситуациях работа будет происходить так, как нам требуется.

С учётом перечисленных примеров можно дать следующее определение полиморфизма. Полиморфизм – особый вид перекрытия методов при наследовании, при котором программный код, работавший с методами родительского класса, пригоден для работы с изменёнными методами дочернего класса.

Вернёмся к иерархии классов `TFigure`, `TCircle`, `TSquare`. В базовом классе `TFigure` реализация методов `Draw` и `Hide` абсолютно не важна, так как они всё равно будут перекрываться в классах-наследниках. Мы оформили реализацию этих методов в виде пустых процедур. Более элегантное решение состоит в объявлении таких методов как *абстрактных*, не нуждающихся в реализации. Абстрактные методы объявляются при помощи директивы `abstract`, указанной после директивы `virtual`:

```
type TFigure = class
       procedure Draw;virtual;abstract;
       procedure Hide;virtual;abstract;
     end;
// теперь реализацию TFigure.Draw и TFigure.Hide писать не надо
```

Директива `abstract` применяется только для виртуальных методов в корневых классах иерархий. Попытка создать экземпляр класса, содержащего абстрактные методы, вызовет предупреждение, а обращение к абстрактному методу сгенерирует исключительную ситуацию.

Полиморфизм позволяет создавать в классах *виртуальные свойства*. Для этого методы чтения и записи свойства объявляются как виртуальные. Определив такие свойства в базовом классе, можно менять их поведение, перекрывая методы чтения и записи в дочерних классах. Виртуальные методы работы со свойствами обычно помещают в секцию `protected`.

Рассмотрим, как связан полиморфизм с вопросами создания и уничтожения объектов. Предположим, что имеется иерархия классов и планируется использовать набор объектов этих классов, разместив объекты в массиве или списке. Создание объектов в наборе – операция достаточно специфическая, каждый объект создаётся вызовом персонального конструктора. А вот уничтожать созданные

объекты можно было бы и сообща, в одном цикле. Для этого деструктор в иерархии классов должен быть полиморфным. Создатели Object Pascal посчитали данную ситуацию стандартной и объявили деструктор `TObject.Destroy` как виртуальный. Чтобы не обрывать цепочку виртуальных методов, рекомендуется деструкторы `Destroy` в пользовательских классах объявлять без параметров, с директивой `override`.

9. RTTI и размещение объектов в памяти

В Object Pascal после компиляции программы для любого класса сохраняется некая дополнительная информация, которая размещается в памяти непосредственно перед VMT. Эта информация называется *информацией о типе периода времени выполнения* (run-time type information, RTTI). Как было сказано выше, любой объект кроме данных полей содержит указатель на VMT (возможно на пустую таблицу, если у класса и его предков нет виртуальных методов). Следовательно, во время работы программы любой объект может получить доступ к RTTI своего класса. Схема размещения объектов и класса в памяти показана на рис. 1.

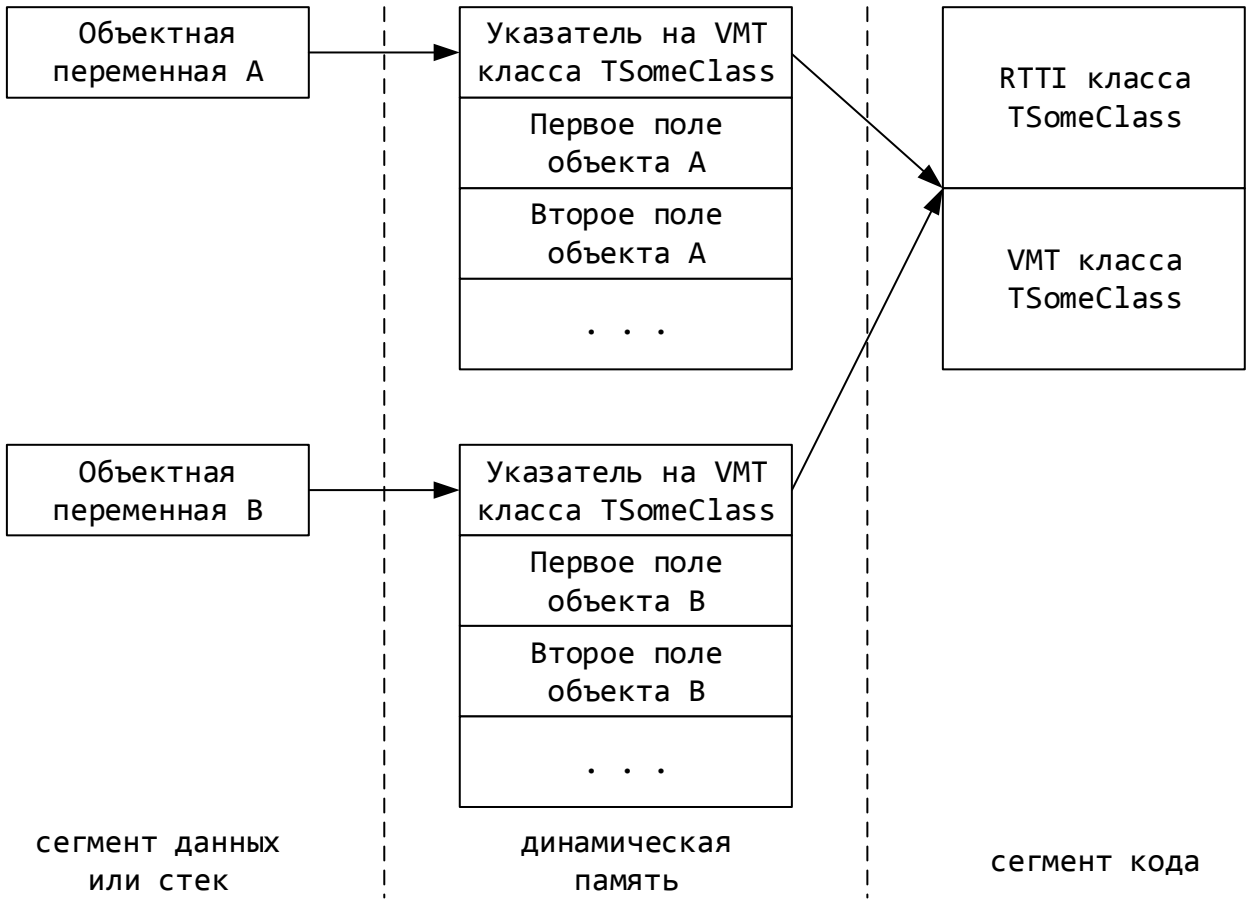

Рис. 1. Схема размещения объектов и RTTI в памяти

Известно, что в RTTI в числе прочих содержатся следующие данные:
1. Указатель на VMT класса-предка;
2. Указатель на строку с именем класса;

3. Размер экземпляра объекта в байтах.

Эти данные позволяют во время выполнения программы *контролировать* (type checking) и *приводить* (type casting) объектные типы.

Для контроля типов используется оператор `is`. Выражение *`объект`* `is` *`класс`* возвращает `true`, если *`объект`* принадлежит указанному классу или потомкам этого класса:

```
if Man is TPerson then . . .
```

Для приведения типов используется оператор `as` в следующей форме:

```
(Man as TPerson).SetAge(37);
```

Допустима традиционная конструкция приведения типов в виде `TPerson(Man).SetAge(37)`, однако оператор `as` является более безопасным. В случае неудачи (то есть, когда объект не относится к указанному классу или его потомкам) он генерирует обрабатываемую исключительную ситуацию, а жёсткое приведение типов может привести к краху приложения.

Литература

1. Архангельский, А. Delphi 2006. Справочное пособие. Язык Delphi, классы, функции Win32 и .NET / А. Я. Архангельский; – СПб. : Бином-Пресс, 2009. – 1152 с. : ил.

2. Кэнту, М. Delphi 2005. Для профессионалов / Марко Кэнту ; пер. с англ. – СПб. : Питер, 2007. – 912 с. : ил.

3. Calvert, Ch. Object Pascal Style Guide / Charles Calvert [Электронный ресурс]. Режим доступа : http://edn.embarcadero.com/article/10280.