Основы объектно-ориентированного программирования

5.1. Три источника и три составные части ООП.

Аббревиатура ООП расшифровывается как объектно-ориентированное программирование. Рассмотрим три источника и три составные части марксизма-ленинизма, ой простите (куда это меня занесло?!), конечно же, ООП!

Три источника – это объекты, абстракция и классификация.

Основная идея ООП заключается в объединении данных, с которыми работает программа и процедур, которые эти данные обрабатывают в единое целое – объект. Такая организация программы позволила максимально приблизить к естественному восприятию человеком окружающих его предметов, сущностей и понятий. Ведь человек воспринимает окружающий его мир, предметы и явления в совокупности свойств, составляющих их элементов и их поведения.

Программист при решении задачи из какой-либо предметной области, выделяет отдельные объекты, исходя из особенностей задачи. Этот процесс называется объектной декомпозицией [5]. Объекты состоят из данных, описывающих свойства этих объектов и процедур, обрабатывающих эти данные. Например, при создании, скажем, базы данных студентов некоторого университета, можно выделить объект "Студент". Данными (свойствами) для этого объекта могут выступать фамилия и имя студента, курс, группа, его оценки и т.д. При этом можно определить некоторые процедуры для обработки этих данных, например процедуру вычисления средней оценки за семестр (эту процедуру можно в дальнейшем использовать для установления размера стипендии), процедуру перевода с курса на курс, процедуру отчисления (к сожалению) из университета и т.д.

При рассмотрении объектов очень важен уровень абстракции (или уровень детализации) с которой мы рассматриваем объект. В нашем случае нас совершенно не интересуют такие характеристики объекта как рост, цвет глаз, размер обуви и т.д. Мы абстрагируемся от этих свойств и выделяем только те свойства, которые позволяют нам решать поставленную задачу.

В то же время важно понимать, что конкретный "экземпляр" студента, например по фамилии Иванов является представителем целого класса студентов. Эту классификацию можно продолжать и развивать как говорится "в разные стороны". Вообще студент, студент какой-либо группы, студент какого-нибудь ВУЗа. Следовательно, мы можем ввести в рассмотрение объект "Группа", объект "ВУЗ". Наконец, студент без всякого сомнения является человеком!! Отсюда мы можем говорить о таком объекте, как "Человек".

Три составные части – это инкапсуляция, наследование и полиморфизм.

Объединение данных и обрабатывающих их функций и процедур в виде отдельных объектов называется инкапсуляцией. Основная сложность ООП заключается в искусстве выделить объекты, используя абстрагирование и классификацию, которые как можно точнее описывали бы решаемую задачу и, кроме того, позволяли бы их повторное использование. Внутреннее "устройство" объекта может быть достаточно сложным, но оно "скрыто от чужих глаз". Для связи с "внешним миром" используются лишь небольшие объемы данных, причем количество и тип этих данных строго под контролем. Это существенно повышает надежность программы.

Наследование одно из важнейших свойств ООП. Создание новых объектов путем использования уже существующих дает программисту ряд преимуществ:

- Не нужно повторно разрабатывать код. Весь код для существующих объектов автоматически может быть использован для новых объектов;
- Вероятность ошибок резко снижается. Если код для уже существующих объектов был уже отлажен и проверен, то любые возникшие ошибки следует искать в кодах, которые были добавлены для новых объектов и, наоборот, если

найдены и исправлены ошибки в кодах для исходных объектов, то они будут автоматически исправлены и в кодах для новых объектов, созданных путем наследования;

- Код программы становится более понятным. Для того чтобы понять как работают объекты, созданные путем наследования достаточно понять как работают добавленные данные и функции.

Под полиморфизмом понимается возможность одних и тех же функций или процедур по разному обрабатывать данные, принадлежащие разным объектам. Например, пусть у вас имеется объект "Геометрическая фигура" и пусть у нее имеется функция вычисления площади. Если необходимо вычислить площадь прямоугольника, то функция будет вычислять ее по одному алгоритму, а если это треугольник, то ее площадь функция будет вычислять по совсем другому алгоритму.

5.2. Классы и объекты.

Основным понятием ООП является класс. Класс представляет собой структуру, состоящую из описания полей данных, функций и процедур, обрабатывающих поля данных и свойств. Функции и процедуры класса принято называть методами. Поля, методы и свойства называются элементами или членами класса.

Поле это обычная переменная языка Паскаль. Допускаются любые разрешенные в языке типы переменных, в том числе поле может иметь тип класса. С помощью полей описываются состояние объекта. С помощью методов – поведение объекта. Свойство это специфический способ для организации связи с данными класса.

Таким образом, класс описывает состояние и поведение объекта. Под объектом понимается конкретный экземпляр (сущность) класса. В программе объ-

ект описывается как переменная типа класс. В языке Free Pascal как и в большинстве современных объектно-ориентированных языков программирования переменная типа класс содержит не сами данные, а ссылку на них, т.е. является указателем, а сам объект размещается в динамической памяти. Однако операция разыменования для классовых переменных в Free Pascal не применяется. Перед использованием объекта необходимо выделить для него память путем создания нового экземпляра класса или присвоением существующего экземпляра переменной типа класс. После завершения работы с этим экземпляром класса (объектом) память необходимо освободить. Объявление класса в простейшем случае производится следующим образом:

```
type
  <Имя класса> = class
  <объявление полей класса>
  <объявление методов класса>
end;
```

Рассмотрим пример объявления класса:

```
type
  THuman = class      // объявление класса
  name: string;   // поля класса
  fam: string;
  function GetData: string; // объявление метода класса
end;
```

Здесь `THuman` имя класса. Имена классов принято начинать с буквы T (от слова `Type`), хотя это и не обязательно. Полями класса являются `name` и `fam`,

а GetData – метод класса (в данном случае это функция).

Реализация метода (тело функции или процедуры) помещается после объявления класса (ключевого слова end). Если класс создается внутри модуля (в большинстве случаев именно так и происходит), то реализацию метода следует помещать после ключевого слова implementation. При этом перед именем метода указывается имя класса через точку:

```
function THuman.GetData: string; // реализация метода
begin
  Result:= name + ' ' + fam;
end;
```

В данном случае мы создали класс THuman (человек) с полями name (имя), fam (фамилия) и методом GetData, который просто возвращает имя и фамилию человека.

Обратите внимание, что все поля класса доступны методам класса и их не надо передавать в качестве формальных параметров. Если вы укажете имена полей класса в качестве формальных параметров метода класса, например таким образом

```
function GetData(name, fam: string): string;
```

то компилятор выдаст ошибку. Точно так же в теле метода нельзя описывать переменные полей класса, например таким образом:

```
function GetData: string;
var
  name: string;
```

5.2.1 Обращение к членам класса.

После того как класс объявлен необходимо создать экземпляр класса путем объявления переменной типа класс:

```
var
  Person: THuman;
```

После описания переменной можно обращаться к членам класса. Обращение производится аналогично обращению к полям записи, т.е. через точку, например:

```
Person.name:= 'Виталий ';
Person.fam:= 'Петров';
fname:= Person.GetData;
```

или с помощью оператора `with`:

```
with Person do
begin
  name:= 'Алексей';
  fam:= 'Морозов';
  fname:= GetData;
end;
```

Напишем программу работы с только что созданным классом:

```
program project1;
{$mode objfpc}{$H+}
```

```
uses
  CRT, FileUtil;
type

  THuman = class      // объявление класса
      name: string;   // поля класса
      fam: string;
      function GetData: string; // объявление метода класса
  end;

function THuman.GetData: string; // реализация метода
begin
  Result:= name + ' ' + fam;
end;

var
  Person: THuman;   // объявление переменной типа класс
  fname: string;

begin
  Person:= THuman.Create; // создание объекта
  Person.name:= 'Виталий'; // присвоение значений полям
  Person.fam:= 'Петров';
  fname:= Person.GetData; // вызов метода
  writeln(UTF8ToConsole('Это: ' + fname));
  with Person do // другой вариант обращения к членам класса
  begin
    name:= 'Алексей';
```

```
    fam:= 'Морозов';
    fname:= GetData;
  end;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;   //  освобождение памяти (уничтожение объекта)
end.
```

В этой программе нами остался не рассмотренным оператор

```
Person:= THuman.Create;
```

С помощью этого оператора в момент выполнения программы объект реально создается (конструируется), т.е. объект размещается в памяти, поля объекта инициализируются необходимыми значениями, указанными в специальном методе, называемым конструктором (о конструкторах мы поговорим позже), адрес этого объекта заносится в `Person` (не забывайте, что это указатель!).

В конце программы память, выделенная объекту, освобождается, т.е. объект уничтожается.

В программе можно использовать сколько угодно экземпляров класса. В следующем примере создается массив объектов. Имена и фамилии вводятся с клавиатуры.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type
```

```
  THuman = class
      name: string;
      fam: string;
      function GetData: string;
      procedure SetData(var f_name, s_name: string);
  end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;

procedure THuman.SetData(var f_name, s_name: string);
begin
  name:= f_name;
  fam:= s_name;
end;

function GetName(var f_name, s_name: string): boolean;
begin
  Result:= true;
  writeln(UTF8ToConsole('Введите имя'));
  readln(f_name);
  if f_name = '***' then
  begin
    Result:= false;
    exit;
  end;
  writeln(UTF8ToConsole('Введите фамилию'));
```

```
  readln(s_name);
end;
const kolHuman = 25;
var
  Person: array [1..kolHuman] of THuman;
  i, kolperson: integer;
  fname, sname: string;
begin
  kolperson:= 0;
  for i:= 1 to kolHuman do
  begin
    if not GetName(fname, sname) then break;
    Person[i]:= THuman.Create;
    Person[i].SetData(fname, sname);
    inc(kolperson);
  end;
  for i:= 1 to kolperson do
  begin
    writeln(UTF8ToConsole('Это: '), Person[i].GetData);
    Person[i].Free;
  end;
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
end.
```

Функция `GetName` не является членом класса. Это обычная функция, которая организует ввод данных. В качестве параметров передаются строковые переменные, в которые вводятся имя и фамилия. После ввода необходимых данных создается новый объект. В класс `THuman` добавлен новый метод

`SetData` с помощью которого введенные данные заносятся в нужные поля класса. Признаком окончания ввода служит строка '***'. Получив такую строку с клавиатуры, функция `GetName` "сигнализирует" вызывающей программе о том, что ввод окончен. При этом новый объект, естественно, создаваться не будет. Переменная `kolperson` это счетчик, которая отслеживает количество реально созданных объектов.

5.3. Инкапсуляция

Выше мы отмечали, что под инкапсуляцией понимается объединение данных и логики их обработки в одно целое – объект. Помимо этого, инкапсуляция подразумевает сокрытие внутренней структуры и ограничение доступа. Связь с "внешним миром" осуществляется через так называемый интерфейс класса, т.е. набором правил, регламентирующих доступ к членам класса. В известном смысле с классом можно ассоциировать некий "черный ящик". Нам известно что он ("черный ящик") может делать. Управлять им мы можем только через его интерфейс. А вот как он это делает нам неизвестно.

Точно так же, нам важно знать что делает и умеет делать класс. А как он это делает нас, вообще говоря, не должно занимать. Этим должен заниматься программист – разработчик класса. Также для нас важно, как этим классом пользоваться (какой у этого класса интерфейс), т.е. какие у этого класса имеются доступные свойства и методы.

Возьмем, для примера, телевизор. Что происходит у него внутри, когда мы его включаем известно "одному аллаху", но нам это и не нужно знать. Возможно, кому-то интересно как устроен телевизор, но для большинства людей важно то, что у телевизора имеются соответствующие кнопки или пульт с кнопками (интерфейс) с помощью которых мы можем смотреть футбол или хоккей (ну, или кому что нравится смотреть).

Принцип сокрытия информации – один из важнейших принципов ООП. Скрывая внутреннее устройство класса, мы абстрагируемся от ненужных деталей. Кроме того, это позволяет защитить члены класса от несанкционированного доступа извне.

В первой программе раздела 5.2.1. мы обращались к членам класса напрямую, например оператором

```
Person.name:= 'Виталий';
```

Вообще говоря, прямое обращение к полям не рекомендуется. Это может стать источником ошибок. Напишем для иллюстрации следующий пример:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type

  THuman = class
      name: string;
      fam: string;
      birthday: integer;
      function GetData: string;
  end;

function THuman.GetData: string;
var
  s: string;
begin
```

```
  str(birthday, s);
  Result:= name + ' ' + fam + ' ' + s;
end;

var
  Person: THuman;

begin
  Person:= THuman.Create;
  writeln(UTF8ToConsole('Введите имя'));
  readln(Person.name);
  writeln(UTF8ToConsole('Введите фамилию'));
  readln(Person.fam);
  writeln(UTF8ToConsole('Введите год рождения'));
  readln(Person.birthday);
  writeln(UTF8ToConsole('Это: '), Person.GetData);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

В класс `THuman` мы добавили новое поле `birthday` (год рождения). Кроме того, немного видоизменили метод `GetData`. Поскольку эта функция возвращает строку, необходимо содержимое поля `birthday`, имеющий тип `integer`, преобразовать в строку. Теперь предположим, при вводе года рождения был введен недопустимый символ. Ваша программа аварийно завершится. Поэтому для изменения значения поля лучше использовать метод (процедуру или функцию). Во второй программе раздела 5.2.1. с массивом объектов мы собственно так и поступили. В методе вы можете организовать контроль над

вводимыми данными. Перепишем предыдущую программу следующим образом:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type

  THuman = class
      name: string;
      fam: string;
      birthday: integer;
      function GetData: string;
      procedure SetData(var f_name, s_name,
                        b_year: string);
  end;

function THuman.GetData: string;
var
  s: string;
begin
  str(birthday, s);
  Result:= name + ' ' + fam + ' ' + s;
end;

procedure THuman.SetData(var f_name, s_name,
                         b_year: string);
var
```

```
  code, year: integer;
begin
  name:= f_name;
  fam:= s_name;
  val(b_year, year, code);
  if code = 0 then
    birthday:= year;
end;

var
  Person: THuman;
  fname, sname, year: string;
begin
  Person:= THuman.Create;
  writeln(UTF8ToConsole('Введите имя'));
  readln(fname);
  writeln(UTF8ToConsole('Введите фамилию'));
  readln(sname);
  writeln(UTF8ToConsole('Введите год рождения'));
  readln(year);
  Person.SetData(fname, sname, year);
  writeln(UTF8ToConsole('Это: '), Person.GetData);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

Здесь мы вводим данные в некоторые вспомогательные переменные. Лишь после окончания ввода методом `SetData` записываем значения в поля объекта

`Person`. Причем в `SetData` осуществляем контроль значения года рождения путем преобразования строки в число функцией `val`. Как вам уже известно, эта функция возвращает ненулевое значение в параметре `code`, если строка не может быть преобразована в число.

Однако, в большинстве случаев, целесообразно вообще запретить доступ к некоторым членам класса извне. Для этого применяются спецификаторы доступа.

5.3.1 Спецификаторы доступа.

Для реализации инкапсуляции имеются следующие спецификаторы (директивы), управляющие видимостью (доступностью) членов класса:

- `private` (частный, говорят еще приватный) – поля и методы класса недоступны из других модулей. Это позволяет полностью скрыть всю "кухню" реализации класса. Однако они доступны в пределах того модуля, где описан данный класс. Более того, если в одном модуле определены несколько классов, то они "видят" приватные разделы друг друга. Это сделано для удобства разработчика данного класса (классов) в этом модуле. Согласитесь, глупо ограничивать в доступе к "внутренностям" телевизора самого изготовителя.
- `protected` (защищенный) – поля и методы класса имеют ограниченную видимость. Они видны в самом классе, во всех классах наследниках этого класса (том числе и в других модулях) и в программном коде, расположенном в том же модуле, что и данный класс.
- `public` (публичный) – свободно доступны из любого места программы, в том числе и из других модулей.

Как правило, поля класса объявляются как `private`, а методы – `public`. Хотя те методы, которые нужны только для внутреннего использования, вполне можно поместить в раздел `private` или `protected`.

Напишем следующую программу:

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil;
type

  THuman = class
    private
      name: string;
      fam: string;
    public
      function GetData: string;
  end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;

var
  Person: THuman;
  fname: string;

begin
  Person:= THuman.Create;
  Person.name:= 'Виталий';
  Person.fam:= 'Петров';
```

```
  fname:= Person.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

Эта программа отличается от первой программы раздела 5.2.1. тем, что мы ввели в описание класса спецификаторы доступа. Здесь поля `name` и `fam` по-прежнему останутся доступны программе, несмотря на то, что они помещены в раздел `private`.

Теперь создайте модуль (меню Файл->Создать модуль) и переместите описание класса в созданный модуль (раздел `interface`), а реализацию метода `GetData` в раздел `implementation`. Остальной код программы оставьте без изменений. Обратите внимание, Lazarus автоматически добавил в объявление `uses` имя вновь созданного модуля.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type

  THuman = class
    private
      name: string;
      fam: string;
    public
```

```
      function GetData: string;
  end;

implementation

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname: string;

begin
  Person:= THuman.Create;
  Person.name:= 'Виталий';
  Person.fam:= 'Петров';
  fname:= Person.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

При попытке компиляции компилятор выдаст ошибку на операторах:

```
Person.name:= 'Виталий';
Person.fam:= 'Петров';
```

Для записи значений в поля `name` и `fam` необходимо создать метод и поместить его объявление в раздел `public`.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  { THuman }
  THuman = class
    private
      name: string;
      fam: string;
    public
      function GetData: string;
      procedure SetData(const f_name, s_name: string);
  end;
implementation

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
```

```
procedure THuman.SetData(const f_name, s_name: string);
begin
  name:= f_name;
  fam:= s_name;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname: string;
begin
  Person:= THuman.Create;
  Person.SetData('Виталий', 'Петров');
  fname:= Person.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

При разработке классов используйте функцию "Автозавершение кода". После того, как вы написали объявление метода, нажмите `Ctrl+Shift+C` и редактор исходного кода Lazarus автоматически сформирует заготовку тела вашего метода. Например, в описании класса наберите

```
procedure SetData(const f_name, s_name: string);
```

и нажмите Ctrl+Shift+C. Lazarus автоматически создаст следующий код:

```
procedure THuman.SetData(const f_name, s_name: string);
begin

end;
```

5.3.2 Свойства.

В предыдущей программе для доступа к частным (private) полям мы использовали методы, которые поместили в раздел public. Фактически мы организовали обмен данными между классом и внешней средой. Но для доступа к полям данных в классе лучше использовать свойства. Свойства описывают состояние объекта, поскольку определяются полями класса и снабжены двумя специальными методами для записи и чтения значения поля. Таким образом, свойство это специальный механизм для организации доступа к полю. Определяется тремя составляющими: поле, метод чтения, метод записи. Преимущество свойства в том, что методы чтения и записи полностью скрыты. Это позволяет вносить изменения в код класса не изменяя использующий его внешний код.

Объявление свойства имеет вид:

property имя 1: тип <read имя 2> <write имя 3>;

где property (свойство) – ключевое слово;

имя 1 – имя свойства, любой разрешенный идентификатор ;

тип – тип поля;

имя 2 – имя метода чтения или непосредственно имя поля;

имя 3 – имя метода записи или непосредственно имя поля;

Угловые скобки означают, что данный параметр может отсутствовать. В случае отсутствия параметра `write` свойство становится свойством "только для чтения". Изменить значение свойства тогда будет невозможно. Отсутствие параметра `read` приводит к тому, что свойство становится свойством "только для записи". Однако это лишено смысла, поскольку значение этого свойства (для чтения) будет недоступно.

Наиболее логично использовать свойство со следующими параметрами:

`property` имя свойства: тип `read` имя поля `write` имя метода записи;

В этом случае для обращения к значению свойства программа производит чтение непосредственно защищенного поля, а для записи значения свойства будет вызываться метод записи. В методе перед записью можно организовать контроль данных на соответствие каким-либо критериям в зависимости от решаемой задачи.

Объявления свойств следует располагать в разделе `public`. При записи свойства вы также можете воспользоваться функцией "Автозавершение кода" редактора исходного кода. Введите

`property` имя свойства: тип

и нажмите `Ctrl+Shift+C`. Lazarus автоматически завершит объявление свойства, а также заготовку тела метода записи. Кроме того, добавит в секцию `private` поле с именем <Fимя свойства> с соответствующим типом и процедуру записи со стандартным именем <Setимя свойства>. Имена полей принято начинать с буквы F, поэтому Lazarus формирует такое имя. Но это не обязательно. Если вас не устраивают имена, предложенные Lazarus, вы можете просто их переименовать. Например, начните набирать описание класса:

```
type
  THuman = class
    private
      name: string;
    public
      property name: string
  end;
```

Нажмите Ctrl+Shift+C. Автоматически будет создан следующий код:

```
type

{ THuman }

  THuman = class
    private
      Fname: string;
      name: string;
      procedure Setname(const AValue: string);
    public
      property name: string read Fname write Setname;
  end;

implementation
procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:= AValue;
```

```
end;
```

Поскольку при использовании функции "Автозавершение кода" Lazarus всегда добавляет новое поле в секцию `private`, логичнее начинать описание класса со спецификатора `public` и определения свойств. В этом случае секция `private` также будет автоматически создана и вы получите описание полей и свойств и готовые шаблоны реализации методов записи. В данном случае мы получили описание поля `Fname` и заготовку процедуры `Setname`.

При использовании свойства нет необходимости явно вызывать процедуру записи. Достаточно записать:

```
Person:= THuman.Create;
Person.name:=  AValue;
```

Таким образом, для пользователя класса свойство будет выглядеть как обычное поле.

Если вы хотите (при использовании функции "Автозавершение кода") для чтения также использовать метод, то вам достаточно перед именем поля в описании свойства добавить слово `Get` и вновь нажать на комбинацию клавиш `Ctrl+Shift+C`. Вы получите заготовку процедуры чтения, например:

```
property name: string read GetFname write Setname;
……
implementation
function THuman.GetFname: string;
begin

end;
```

Рассмотрим примеры работы со свойствами.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  { THuman }
  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setfam(const AValue: string);
      procedure Setname(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
   end;

implementation
{ THuman }
procedure THuman.Setname(const AValue: string);
begin
  if Fname = AValue then exit;
  Fname:= AValue;
end;
procedure THuman.Setfam(const AValue: string);
begin
  if Ffam = AValue then exit;
```

```
  Ffam:= AValue;
end;
function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname: string;
begin
  Person:= THuman.Create;
  Person.name:= 'Виталий';
  Person.fam:= 'Петров';
  fname:= Person.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

В этой программе мы определили два свойства `name` и `fam`. Воспользовавшись функцией автозавершения, мы получили методы записи данных в поля. В программе в явном виде мы эти методы не вызывали. Мы записали про-

сто

```
  Person.name:= 'Виталий';
  Person.fam:= 'Петров';
```

Всю остальную работу (вызов методов записи) за нас выполнил компилятор. Обратите внимание, по сравнению с предыдущей программой `name` и `fam` это не имена полей, а имена свойств. В следующем примере данные считываются с клавиатуры. Добавлен контроль ввода данных.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  { THuman }

  THuman = class
    private
      Fname: string;
      Ffam: string;
      Fbirthday: integer;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
      procedure Setbirthday(const AValue: integer);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
```

```
      property birthday: integer read Fbirthday write
                                        Setbirthday;
      function GetData: string;
   end;

implementation

{ THuman }

procedure THuman.Setname(const AValue: string);
begin
  if Fname = AValue then exit;
  Fname:= AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam = AValue then exit;
  Ffam:= AValue;
end;

procedure THuman.Setbirthday(const AValue: integer);
begin
  if Fbirthday = AValue then exit;
  Fbirthday:= AValue;
end;
function THuman.GetData: string;
var
  s: string;
```

```
begin
  str(birthday, s);
  Result:= name + ' ' + fam + ' ' + s;
end;
end.

program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;
var
  Person: THuman;
  fname, sname, s_year: string;
  code, year: integer;
begin
  Person:= THuman.Create;
  writeln(UTF8ToConsole('Введите имя'));
  readln(fname);
  Person.name:= fname;
  writeln(UTF8ToConsole('Введите фамилию'));
  readln(sname);
  Person.fam:= sname;
  writeln(UTF8ToConsole('Введите год рождения'));
  readln(s_year);
  val(s_year,   year,   code);        //   контроль   ввода
```

```
  if code = 0 then
    Person.birthday:= year
  else
    Person.birthday:= 0;
  fname:= Person.GetData;
  writeln(UTF8ToConsole('Это: '), fname);
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Person.Free;
end.
```

5.4. Наследование

Представьте себе, что имеется класс, который вас в принципе удовлетворяет, но чего-то в нем чуть-чуть не хватает. Что придется заново разрабатывать класс? Конечно же, заново создавать класс не нужно. Достаточно создать класс на основе существующего и просто добавить в него недостающие члены (поля, методы и свойства). Это и есть наследование. Новый класс называется наследником или потомком или дочерним классом. А старый класс называется родительским классом или предком или базовым классом. Объявляется класс наследник следующим образом:

```
type
<Имя класса наследника> = class(Имя класса родителя)
            <Описание полей, методов и свойств,
        характерных только для класса наследника>
end;
<Реализация методов>
```

Если имя класса родителя не указано, то по умолчанию класс наследуется

от самого общего класса TObject. Наш класс THuman из предыдущих примеров является наследником TObject.

Класс наследник получает (наследует) все поля, методы и свойства класса родителя. В то же время созданный класс, в свою очередь может выступать в качестве родителя. В Free Pascal существует так называемая иерархия классов, на вершине которого и находится класс TObject. Таким образом, класс TObject является прародителем всех остальных классов.

Например, создадим класс TStudent (студент). Можно создать этот класс с нуля, но, поскольку студент является человеком, то логично будет создать класс на основе класса THuman.

```
program project1;
{$mode objfpc}{$H+}

uses
  CRT, FileUtil;
type

  THuman = class      // объявление базового класса
    private
      name: string;
      fam: string;
    public
      function GetData: string;
  end;
function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
```

```
type

  TStudent = class(THuman) // объявление класса – наследника
    private
      group: string;
  end;

var

  Student: TStudent;
  fname: string;

begin
  Student:= TStudent.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= 'ПОВТАС-1/09';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
end.
```

Здесь в класс `TStudent` мы ввели поле `group`, характерное только для студента. В данном случае это группа. В то же время, благодаря наследованию, будут доступны поля `name` (имя) и `fam` (фамилия), а также метод `GetData` родительского класса. И класс `THuman` и класс `TStudent` являются потомками класса `TObject`, поэтому для класса `TStudent` доступен метод (конструк-

тор) `Create` класса `TObject` (более подробно о конструкторах смотрите ниже). Обратите внимание, если вы собираетесь использовать только объект класса `TStudent`, то создавать экземпляр класса `THuman` вовсе не обязательно. Более того, вы можете создавать класс наследник в другом модуле, нежели где описан базовый класс.

```
unit Unit1;
{$mode objfpc}{$H+}
interface

uses
  Classes, SysUtils;

type

  THuman = class      // объявление класса
    protected
      name: string;
      fam: string;
    public
      function GetData: string;
  end;
implementation

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.
```

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type

  TStudent = class(THuman) // объявление класса - наследника
    private
      group: string;
  end;

var

  Student: TStudent;
  fname: string;

begin
  Student:=TStudent.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= 'ПОВТАС-1/09';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
end.
```

Здесь мы описали родительский класс в отдельном модуле `Unit1`. Обратите внимание, чтобы поля родительского класса были доступны классу на-

следнику, мы в классе `THuman` объявили поля со спецификатором `protected`.

Если поля родительского класса защитить спецификатором `private`, то дочерний класс может получить доступ к его полям с помощью его свойств (разумеется, в родительском классе должны быть описаны соответствующие свойства).

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;
type

  { THuman }
  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
   end;
```

```
implementation

{ THuman }

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.
program project1;
{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;

type
```

```
  TStudent = class(THuman) //  объявление класса - наследника
    private
       group: string;
  end;

var
  Student: TStudent;
  fname: string;

begin
  Student:= TStudent.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= 'ПОВТАС-1/09';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
end.
```

Напишем класс `TProfessor` (преподаватель). Преподаватель тоже является человеком (или у вас есть сомнения на этот счет?!), поэтому совершенно естественно, что этот класс будет также наследоваться от класса `THuman`. Так же как студенты "кучкуются" в группы, так и преподаватели объединяются в кафедры. Поэтому для класса введем поле `kafedra`. В следующем примере создаются сразу два класса наследника. Приведу только код основной программы. Класс `THuman` (модуль `Unit1`, см. предыдущий пример) не претерпит никаких изменений.

```
program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type
  TStudent = class(THuman) // объявление класса - наследника
    private
      group: string;
  end;
  type
    TProfessor = class(THuman) // объявление класса - наследника
      private
        kafedra: string;
    end;
var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;
begin
  Student:= TStudent.Create;
  Professor:= TProfessor.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= 'ПОВТАС-1/09';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor.name:= 'Иван';
  Professor.fam:= 'Иванов';
  Professor.kafedra:='Программирование';
```

```
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
end.
```

5.5. Полиморфизм

Во всех примерах раздела 5.4. мы использовали метод `GetData` родительского класса. Но он возвращает только значения полей `name` и `fam`. Поэтому, несмотря на то, что мы в программе указывали значения полей `group` и `kafedra`, они на экран не выводились. Как же вывести значения этих полей?

Решение заключается в написании соответствующего метода в дочернем классе. Причем мы вольны присваивать этому методу любое имя. Например, мы можем написать методы:

```
function TStudent.A: string;
begin
  Result:= name + ' ' + fam + ', группа ' + group;
end;

function TProfessor.B: string;
begin
  Result:= name + ' ' + fam + ', кафедра ' + kafedra;
end;
```

Но, давайте вспомним, что методы как мы отмечали выше, характеризуют некоторые действия по обработке данных класса. В этом контексте мы можем ввести в рассмотрение действие "Получить сведения об объекте". В одном случае это будет "Получить сведения о человеке", в другом случае "Получить сведения о студенте", в третьем – "Получить сведения о преподавателе". Во всех трех случаях это в принципе однотипные действия, хотя и несколько различающиеся по сути. Мы можем написать совсем коротко – "Получить сведения".

Так вот, возвращаясь "к нашим баранам", мы можем констатировать, что функция `GetData` это и есть действие "Получить сведения". Поэтому мы можем использовать в дочерних классах методы с таким же именем, что и в родительском классе. Конечно, реализации этих методов различаются и даже могут различаться кардинально, но по сути, если вы хотите реализовать однотипные аспекты поведения объектов, вы можете и должны давать одинаковые имена их методам. Давая им различные имена, вы можете очень скоро запутаться, особенно если производных классов достаточно много. Отсюда, мы можем написать:

```
function TStudent.GetData: string;
begin
  Result:= name + ' ' + fam + ', группа ' + group;
end;

function TProfessor.GetData: string;
begin
  Result:= name + ' ' + fam + ', кафедра ' + kafedra;
end;
```

Такое явление, когда методы класса родителя и методы дочерних классов имеют одинаковые имена, но различные реализации называется полиморфизмом. Функции `THuman.GetData`, `TStudent.GetData` и `TProfessor.GetData` несомненно различаются, поскольку возвращают

строки с разным содержанием. В одном случае просто имя и фамилию, в другом имя, фамилию и наименование группы, а в третьем случае – имя, фамилию и название кафедры. Можно еще более подчеркнуть их различие, если в классе `TStudent` описать тип поля `group` как `integer`. В этом случае будет возвращаться не название группы, а его номер. Реализация этого метода будет следующая:

```
function TStudent.GetData: string;
var
  s: string;
begin
  str(group, s);
  Result:= name + ' ' + fam + ', группа ' + s;
end;
```

5.5.1 Раннее связывание.

Одноименные методы, определенные таким образом, называются статическими. При вызове метода, помимо явно описанных параметров функции или процедуры, неявно передается еще один параметр `self` – указатель на объект, вызвавший метод. Таким образом, компилятор легко определяет объект какого класса вызвал метод и организует вызов нужного метода. Это так называемое раннее связывание. Метод дочернего класса как бы подменяет метод родительского класса с тем же именем. Гораздо чаще применяются термины перекрытие или переопределение. Можно было сказать – метод дочернего класса перекрывает (переопределяет) метод родительского класса с тем же именем. При этом количество и типы параметров нового метода дочернего класса и переопределяемого метода могут не совпадать. Рассмотрим пример, где используется механизм раннего связывания.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type

  { THuman }

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
   end;
implementation

{ THuman }

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
  Fname:=AValue;
end;
```

```
procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam;
end;
end.

program project1;

{$mode objfpc}{$H+}

uses
  CRT, FileUtil, Unit1;

type

  TStudent = class(THuman)
    private
      group: string;
    public
      function GetData: string;
  end;

function TStudent.GetData: string;
```

```
begin
  Result:= name + ' ' + fam + ', группа ' + group;
end;

  type

    TProfessor = class(THuman)
      private
        kafedra: string;
      public
        function GetData: string;
    end;

function TProfessor.GetData: string;
begin
  Result:= name + ' ' + fam + ', кафедра ' + kafedra;
end;

var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;

begin
  Student:= TStudent.Create;
  Professor:= TProfessor.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= '"ПОВТАС-1/09"';
```

```
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor.name:= 'Иван';
  Professor.fam:= 'Иванов';
  Professor.kafedra:='"Программирование"';
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
  Professor.Free;
end.
```

Посмотрите на реализации методов `GetData`. В чем-то они похожи. Особенно если мы перепишем их в виде:

```
function TStudent.GetData: string;
begin
  Result:= name + ' ' + fam;
  Result:= Result + ', группа ' + group;
end;
function TProfessor.GetData: string;
begin
  Result:= name + ' ' + fam;
  Result:= Result + ', кафедра ' + kafedra;
end;
```

Первые операторы этих функций совпадают с методом родительского класса. Это в наших примерах методы очень простые. В реальности методы могут быть очень сложными и насчитывать не один десяток, а то и сотен строк

кода. Что, надо копировать весь похожий код в каждый из дочерних методов? Конечно же, нет! Оказывается можно вызывать метод родительского класса с помощью ключевого слова `inherited`. Например:

```
function TStudent.GetData: string;
begin
  Result:= inherited GetData;
  Result:= Result + ', группа ' + group;
end;
```

Здесь сначала вызывается метод родительского класса, а затем добавляется новый код.

5.5.2 Позднее связывание.

Выше мы видели, что для статических методов разрешение связей, т.е. определение того, какой именно метод следует вызывать, происходит на этапе компиляции. Но бывают ситуации, когда компилятор не может определить, к какому объекту (экземпляру класса) относится метод. Рассмотрим пример. Пусть нам необходимо, чтобы на экран выводилась кроме прежней информации еще и статус человека, т.е. кто он – студент или преподаватель. Для этого введем новый метод `Status`. Этот метод будет вызываться из метода `GetData`. Во время компиляции неизвестно, объект какого класса вызывает метод `Status`. Это определяется на этапе выполнения, когда точно известен активный в данный момент объект. Разрешение связей на этапе выполнения называется поздним связыванием. Для реализации механизма позднего связывания применяются ключевые слова `virtual` и `override`. Метод родительского класса помечается ключевым словом `virtual`. Методы дочерних классов, перекрывающих метод родительского класса помечаются ключевым словом

override, а все эти методы, включая и родительский, называются виртуальными методами. Причем в отличие от статических методов, количество, тип и порядок следования параметров в виртуальных методах должны совпадать. Рассмотрим пример.

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils;
type
  { THuman }
  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      function GetData: string;
      function Status: string; virtual;
   end;
implementation
{ THuman }
procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
```

```
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.Status: string;
begin

end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam + Status;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type

  TStudent = class(THuman)
    private
      group: string;
```

```
    public
      function GetData: string;
      function Status: string; override;
  end;

function TStudent.Status: string;
begin
  Result:= ' - студент';
end;
function TStudent.GetData: string;
begin
  Result:= inherited GetData + ', группа ' + group;
end;
type
  TProfessor = class(THuman)
    private
      kafedra: string;
    public
      function GetData: string;
      function Status: string; override;
  end;

function TProfessor.Status: string;
begin
  Result:= ' - преподаватель';
end;

function TProfessor.GetData: string;
begin
```

```
  Result:= inherited GetData + ', кафедра ' + kafedra;
end;

var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;
begin
  Student:= TStudent.Create;
  Professor:= TProfessor.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= '"ПОВТАС-1/09"';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor.name:= 'Иван';
  Professor.fam:= 'Иванов';
  Professor.kafedra:= '"Программирование"';
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
  Professor.Free;
end.
```

Кроме виртуальных методов могут быть объявлены и так называемые динамические методы. Они объявляются с помощью ключевого слова `dynamic`.

А в классах наследниках все того же ключевого слова `override`. С точки зрения наследования, методы этих двух видов одинаковы: они могут быть перекрыты в дочернем классе только одноименными методами, имеющими тот же тип. Разница заключается в реализации механизма позднего связывания. Для виртуальных методов используется специальная таблица виртуальных методов (ТВМ), а для динамических методов таблица динамических методов (ТДМ).

В ТВМ хранятся адреса всех виртуальных методов класса, включая и родительских классов, даже если они не перекрыты в данном классе. Поэтому вызовы виртуальных методов происходят достаточно быстро, однако ТВМ требует больше памяти.

С другой стороны, каждому динамическому методу присваивается уникальный индекс. В ТДМ хранятся индексы и адреса только тех динамических методов, которые описаны в данном классе, поэтому ТДМ занимает меньше памяти. При вызове динамического метода, происходит поиск в ТДМ данного класса. Если поиск не дал результатов, поиск продолжается в ТДМ всех классов-родителей в порядке иерархии. Чем больше глубина иерархии классов, тем медленнее будут работать вызовы динамических методов.

Чтобы реализовать динамические методы для нашего примера, достаточно заменить слово `virtual`, на `dynamic`.

Посмотрим теперь на код метода `Status` в родительском классе. Фактически в этом методе ничего не делается, функция возвращает пустую строку. В таких случаях удобнее объявить метод как абстрактный с помощью ключевого слова `abstract`. Реализация метода, который объявлен как абстрактный, в данном классе не производится, но обязательно должен быть переопределен в дочерних классах. Обратите внимание, только виртуальные методы могут быть объявлены абстрактными. Для нашего примера чтобы объявить абстрактный метод добавьте в классе `THuman` в объявление метода `Status` после слова `virtual` через точку с запятой ключевое слово `abstract` и удалите код реализации метода.

5.5.3 Конструкторы и деструкторы.

Для создания объектов мы применяли метод `Create`. Это так называемый конструктор, т.е. специальный метод класса, который предназначен для выделения памяти и размещения в памяти экземпляра класса. Кроме того, в задачу конструктора входит инициализация значений полей экземпляра класса. Стандартный конструктор устанавливает все данные нового экземпляра класса в ноль. В результате все числовые поля и поля порядкового типа приобретают нулевые значения, строковые поля становятся пустыми, а поля, содержащие указатели и объекты получают значение `nil`. Под стандартным конструктором имеется в виду конструктор, унаследованный классом от класса `TObject`.

Если нужно инициализировать данные экземпляра класса определенными значениями, то необходимо написать свой собственный конструктор. Допускается использование нескольких конструкторов. Желательно, чтобы все конструкторы имели стандартное имя `Create`. Это позволяет переопределять конструкторы, включая и стандартный конструктор, для выполнения каких-либо полезных действий. Описание конструктора производится с помощью ключевого слова `constructor`. Давайте в последнем примере предыдущего раздела в класс `THuman` добавим конструктор. Код программы будет следующим:

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;
type
```

```
  { THuman }

  THuman = class
    private
      Fname: string;
      Ffam: string;
      procedure Setname(const AValue: string);
      procedure Setfam(const AValue: string);
    public
      property name: string read Fname write Setname;
      property fam: string read Ffam write Setfam;
      constructor Create; // конструктор
      function GetData: string;
      function Status: string; virtual; abstract;
   end;

implementation

{ THuman }

constructor THuman.Create; // реализация конструктора
begin
  Fname:= 'Андрей';
  Ffam:= 'Аршавин';
end;

procedure THuman.Setname(const AValue: string);
begin
  if Fname=AValue then exit;
```

```
  Fname:=AValue;
end;

procedure THuman.Setfam(const AValue: string);
begin
  if Ffam=AValue then exit;
  Ffam:=AValue;
end;

function THuman.GetData: string;
begin
  Result:= name + ' ' + fam + Status;
end;
end.

program project1;
{$mode objfpc}{$H+}
uses
  CRT, FileUtil, Unit1;
type

  TStudent = class(THuman)
    private
      group: string;
    public
      function GetData: string;
      function Status: string; override;
  end;
```

```
function TStudent.Status: string;
begin
  Result:= ' - студент';
end;

function TStudent.GetData: string;
begin
  Result:= inherited GetData + ', группа ' + group;
end;

type

  TProfessor = class(THuman)
    private
      kafedra: string;
    public
      function GetData: string;
      function Status: string; override;
  end;

function TProfessor.Status: string;
begin
  Result:= ' - преподаватель';
end;

function TProfessor.GetData: string;
begin
  Result:= inherited GetData + ', кафедра ' + kafedra;
end;
```

```
var
  Student: TStudent;
  Professor: TProfessor;
  fname: string;
begin
  Student:= TStudent.Create;
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor:= TProfessor.Create;
  Student.name:= 'Виталий';
  Student.fam:= 'Петров';
  Student.group:= '"ПОВТАС-1/09"';
  fname:= Student.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  Professor.name:= 'Иван';
  Professor.fam:= 'Иванов';
  Professor.kafedra:= '"Программирование"';
  fname:= Professor.GetData;
  writeln(UTF8ToConsole('Это: ' + fname));
  writeln(UTF8ToConsole('Нажмите любую клавишу'));
  readkey;
  Student.Free;
  Professor.Free;
end.
```

При выполнении оператора

```
Student:= TStudent.Create;
```

сработает конструктор, определенный нами в классе `THuman`, поскольку он пе-

рекрывает стандартный конструктор. В результате поля `Fname` и `Ffam` будут инициализированы значениями `'Андрей'` и `'Аршавин'`. После выполнения операторов

```
fname:= Student.GetData;
writeln(UTF8ToConsole('Это: ' + fname));
```

на экран будет выведено
"Это: Андрей Аршавин – студент, группа".

Обратите внимание, название группы выведено не будет. Для инициализации значения поля `group` необходимо написать конструктор в классе `TStudent`, например таким образом:

```
constructor TStudent.Create;
begin
  group:= '"Арсенал"';
end;
```

Теперь при выполнении оператора

```
Student:= TStudent.Create;
```

будет вызван конструктор, определенный в классе `TStudent` и на экран будет выведено:

"Это: – студент, группа "Арсенал"".

Мы видим, что имя и фамилия отсутствуют. Это опять происходит из-за того, что конструктор `Create` дочернего класса `TStudent` перекрывает конструк-

тор родительского класса THuman. Чтобы вызвать конструктор родительского класса, необходимо использовать ключевое слово inherited. Таким образом, реализацию конструктора класса TStudent необходимо записать в виде:

```
constructor TStudent.Create;
begin
  inherited Create;
  group:= '"Арсенал"';
end;
```

Теперь на экран будет выведена строка так, как мы хотели:
"Это: Андрей Аршавин – студент, группа "Арсенал"".

Вполне возможно использовать конструкторы с параметрами. Вот как, например, может выглядеть конструктор с параметрами класса THuman:

```
public
      constructor Create(n, f: string);
……
end;
implementation
{ THuman }
constructor THuman.Create(n, f: string);
begin
  Fname:= n;
  Ffam:= f;
end;
```

Конструктор класса TStudent:

```
public
      constructor Create(gr: string);
……
end;
constructor TStudent.Create(gr: string);
begin
  inherited Create('Андрей', 'Аршавин');
  group:= gr;
end;
```

Если вы теперь вызовете конструктор в виде

```
Student:= TStudent.Create;
```

то компилятор будет "сильно ругаться"! Необходимо вызывать конструктор таким образом:

```
Student:= TStudent.Create('"Арсенал"');
```

Обратите внимание и на вызов конструктора родительского класса

```
inherited Create('Андрей', 'Аршавин');
```

Поскольку, мы с вами отлично знаем, что Андрей Аршавин не студент, видоизмените описания классов, чтобы на экран выводилось, ну что-то типа:
"Андрей Аршавин – футболист команды Арсенал".

В заключение отметим, что конструктор создаёт новый объект только в том случае, если перед его именем указано имя класса. Если указать имя уже существующего объекта, он поведёт себя по-другому: не создаст новый объект, а только выполнит код, содержащийся в теле конструктора.

Деструктор имеет стандартное имя `Destroy` и предназначен для уничто-

жения объекта:

```
Student.Destroy;
```

В теле деструктора обычно должны уничтожаться встроенные объекты и динамические данные, как правило, созданные конструктором. Как и обычные методы, деструктор может иметь параметры, но эта возможность используется крайне редко. Для объявления деструктора используется ключевое слово `destructor`.

`Destroy` – это виртуальный деструктор класса `TObject`. Поэтому при переопределении деструктора его необходимо объявлять с ключевым словом `override`, например:

```
destructor Destroy; override;
```

В теле самого деструктора необходимо вызывать родительский деструктор (`inherited Destroy`). Этим обеспечивается формирование цепочки деструкторов, восходящей к деструктору класса `TObject`, который и осуществляет освобождение памяти, занятой объектом.

Рекомендуется вместо метода `Destroy` использовать метод `Free`. Этот метод вначале проверяет, не является ли текущий объект `nil` и лишь, затем вызывает метод `Destroy`. Если объекты создаются в начале работы программы, а уничтожаются в самом конце, то применение метода `Free` является самым оптимальным.

На этом мы закончим наш краткий экскурс в объектно-ориентированное программирование. В следующей главе мы еще коснемся некоторых аспектов ООП применительно к созданию программ с графическим интерфейсом.

Более подробные сведения о ООП вы можете почерпнуть из [5] и [6]. Хотя

–

в этих книгах материал излагается применительно к Turbo Pascal и Delphi, тем не менее, вы можете использовать их, особенно [6].

Если вы помните, при создании нашего самого первого консольного приложения (см. 2.1.10) Lazarus предложил нам заготовку кода, который мы с вами просто удалили. Тогда нам этот код был совершенно непонятен. Ну а теперь…, теперь вы можете без труда разобраться с этим кодом! Да-да, Lazarus создал для нас заготовку класса с конструктором и деструктором!