Краткое введение в современный Object Pascal для программистов

Michalis Kamburelis

Table of Contents

1. Введение: Для чего это нужно?

В мире существует множество книг о Паскале, однако, многие из них говорят об "устаревшем" Паскале, без классов, модулей, generic-ов (дженериков) и многих других современных методов и приёмов программирования.

С целью заполнения этой бреши и было написано это краткое введение о том, что часто называется **современным объектным Паскалем**. Чаще его называют просто *"Паскаль"* или *"Наш Паскаль"*. Впрочем, представляя язык, важно подчеркнуть, что это современный, объектно-ориентированный язык, который был **существенно** усовершенствован по сравнению со старым (Turbo) Pascal, который когда-то давно изучали в школах и институтах. Сегодня его вполне можно сравнивать с такими известными языками программирования, как C++, Java или C#.

- Паскаль имеет все современные особенности, которые можно ожидать -- классы, модульную структуру, интерфейсы, generic-и…
- Паскаль компилируется в быстрый машинный код.
- Паскаль является типобезопасным языком, что в некоторых случаях существенно упрощает отладку.
- В основном Паскаль - высокоуровневый язызк, однако он имеет возможность использовать низкоуровневые подходы если это необходимо.

Он так же имеет превосходный кроссплатформенный портативный компилятор с открытым исходным кодом: *Free Pascal Compiler* (http://freepascal.org/). Для него существует несколько IDE (включающих в себя редактор, отладчик, библиотеки компонентов, дизайнер форм), одна из наиболее известных называется *Lazarus* (http://lazarus.freepascal.org/). Автора данной книги является создателем *Castle Game Engine* (https://castle-engine.io/) - который является мощным портативным двух- и трёхмерным игровым движком, использующим этот язык, для создания игр под многие платформы: Windows, Linux, MacOSX, Android, iOS, web плагины.

Данное краткое введение в первую очередь ориентировано на программистов, которые уже имеют некоторый опыт в других языках программирования. Здесь не будет раскрываться значение некоторых универсальных концепций, таких как *"что такое класс"*, лишь будет показано как они реализуются в современном Паскале.

Прим.перев. Здесь и далее мы будем использовать оригинальные английские понятия такие как unit, constructor, override, reference-counting в случаях, если они являются ключевыми словами языка либо не имеют общепринятых переводов на русский язык. Мы также часто предпочитаем использовать оригинальное английское слово его транслитерации, как в случае с понятием "generic" выше.

2. Основы

2.1. Программа "Hello world!"

По доброй традиции знакомство с языком начинают с самой базовой и простой программы "Hello world". На Паскале она выглядит следующим образом:

```
{$mode objfpc}{$H+}{$J-} // Эту строку необходимо использовать во всех
 современных программах

program MyProgram; // Сохраните этот файл под названием myprogram.lpr
begin
  WriteLn('Hello world!');
end.
```

Это — полноценная программа, которую можно *скомпилировать* и *запустить*.

- Это можно сделать с помощью командной строки FPC, просто создав новый файл `myprogram.lpr` с кодом программы внутри и выполнив команду `fpc myprogram.lpr`.
- Если используется *Lazarus*, то необходимо создать новый проект (в строке меню: *Project* → *New Project* → *Simple Program*). Сохраните его как `myprogram` и скопируйте в него этот исходный код. Компиляция выполняется используя пункт *Run* → *Compile* в меню.
- Данная программа является консольной, то есть, в обоих случаях скомпилированный исполняемый файл нужно будет запустить из терминала командной строки, чтобы увидеть результат.

Остальная часть этой книги рассказывает о самом объектном Паскале, поэтому не ожидайте чего-либо большего, чем консольных приложений. Если хочется взглянуть на что-либо более "крутое", можно просто создать новый GUI проект в *Lazarus* (*Project* → *New Project* → *Application*). Вуаля! — рабочее кроссплатформенное GUI приложение, с нативным видом, использующее удобные визуальные библиотеки. *Lazarus* и *Free Pascal Compiler* имеют множество готовых компонент для сетей, GUI, баз данных, чтения и записи различных форматов файлов (XML, json, изображения…), управления потоками и всем, что только может понадобиться программисту. Ярким тому примером является *Castle Game Engine*, о котором упоминалось ранее :)

2.2. Функции, процедуры, простейшие типы

```
{$mode objfpc}{$H+}{$J-}

program MyProgram;

procedure MyProcedure(const A: Integer);
begin
  WriteLn('A + 10 составляет: ', A + 10);
end;

function MyFunction(const S: string): string;
begin
  Result := S + 'строки управляются автоматически';
end;

var
  X: Single;
begin
  WriteLn(MyFunction('примечание: '));
  MyProcedure(5);

  // деление с помощью оператора "/" всегда даёт результат с плавающей
 запятой
  // для целочисленного деления необходимо использовать "div"
  X := 15 / 5;
  WriteLn('X составляет: ', X); // отобразить в научном формате вида
 3.0000000E+000
  WriteLn('X составляет: ', X:1:2); // отобразить 2 знака после запятой
end.
```

Чтобы задать возвращаемое значение функции, необходимо присвоить какое-либо значение "волшебной" переменной `Result` в процессе выполнения функции. Её можно свободно читать и устанавливать новое значение так же просто как и любую локальную переменную.

```
function MyFunction(const S: string): string;
begin
  Result := S + ' Добавим что-нибудь';
  Result := Result + ' и ещё что-нибудь!';
  Result := Result + ' И ещё немножко!';
end;
```

Можно рассматривать имя функции (`MyFunction` в примере выше) как переменную, которую можно использовать как самую обычную переменную. Но лично я бы не советовал так делать, поскольку это выглядит не очень "прозрачно", особенно в случае, если это значение используется с правой стороны выражения. Лучше всегда использовать `Result`, в случае, если необходимо использовать или устанавливать возвращаемое значение функции.

Естественно, при необходимости можно вызывать функцию рекурсивно. Но при этом в случае, если вызывается беспараметрическая функция, следует не забывать указать пустые скобки `()` (которые в обычных случаях в Паскале можно опускать) - иначе будет просто выполнен доступ к результату текущей функции. Например:

```
function ReadIntegersUntilZero: string;
var
  I: Integer;
begin
  Readln(I);
  Result := IntToStr(I);
  if I <> 0 then
    Result := Result + ' ' + ReadIntegersUntilZero();
end;
```

Функция `Exit` служит для завершения выполнения процедуры или функции до того, как она достигнет завершающего `end;`. Если `Exit` вызвать без параметров, она вернёт последнее значение, присвоенное `Result`. Так же можно использовать конструкцию `Exit(X)`, чтобы установить результат функции и завершить её исполнение **немедленно** — это эквивалентно конструкции `return X` в C-подобных языках.

```
function AddName(const ExistingNames, NewName: string): string;
begin
  if ExistingNames = '' then
    Exit(NewName);
  Result := ExistingNames + ', ' + NewName;
end;
```

Ещё одна полезная особенность Паскаля заключается в том, что можно "отбросить" результат функции и использовать её как обычную процедуру. Например:

```
var
  Count: Integer;
  MyCount: Integer;

function CountMe: Integer;
begin
  Inc(Count);
  Result := Count;
end;

begin
  Count := 10;
  CountMe; // результат функции будет отброшен, однако функция выполняется
  MyCount := CountMe; //обычное применение функции
end.
```

2.3. Условные операторы (if)

Конструкции `if .. then` или `if .. then .. else` запускают определённый код, когда выполняется указанное условие. В отличии от C-подобных языков, в Паскале нет строгого требования ставить условие в скобки.

```
var
  A: Integer;
  B: boolean;
begin
  if A > 0 then
    DoSomething;

  if A > 0 then
  begin
    DoSomething;
    AndDoSomethingMore;
  end;

  if A > 10 then
    DoSomething
  else
    DoSomethingElse;

  // идентично предыдущему примеру
  B := A > 10;
  if B then
```

```
    DoSomething
  else
    DoSomethingElse;
end;
```

Оператор `else` всегда относится только к последнему условию `if`. Поэтому можно вполне рассчитывать на однозначность выполнения такого кода:

```
if A <> 0 then
  if B <> 0 then
    AIsNonzeroAndBToo
  else
    AIsNonzeroButBIsZero;
```

Впрочем, заключение вложенного `if` внутри блока `begin … end;` является лучшим вариантом, чем предыдущий пример, поскольку он более читабельный, даже если нарушены отступы или большой объём кода и комментариев затрудняет понимание. Таким образом, всегда очевидно к какому `if` относится данный `else` - относительно A или относительно B - и, соответственно, куда меньше шансов допустить ошибку при написании кода или при его правках.

```
if A <> 0 then
begin
  if B <> 0 then
    AIsNonzeroAndBToo
  else
    AIsNonzeroButBIsZero;
end;
```

2.4. Логические операторы, операторы отношений и побитовые (поразрядные) операторы

Логические операторы представлены в Паскале операторами `and`, `or`, `not`, `xor`. Их значение в большинстве случаев очевидно для людей, знакомых с компьютерной грамотой. Разве что, за исключением оператора `xor`, который в русской литературе обычно называется *"исключающее или"*. Эти операторы принимают *булевские аргументы (boolean)*, и возвращаемое значение имеет тот же тип *boolean*. Они также могут работать как *побитовые операторы*, в случае, если оба аргумента целого типа (integer, byte или подобные), в этом случае они также возвращают значение идентичного целого типа.

Операторы отношения (сравнения) - представлены следующими комбинациями символов: `=`, `<>`, `>`, `<`, `<=`, `>=`, значение которых вполне очевидно. Следует отметить, что в отличии от синтаксиса C-подобных языков, в Паскале оператор сравнения выглядит как один знак "равно" `A = B` (в отличии от C, где используется код `A == B`). Специальным *оператором присваивания* в Паскале является `:=`.

Логический (или побитовый) оператор имеет более высокий приоритет, чем операторы отношения. Поэтому, может понадобиться использовать круглые скобки вокруг сравниваемых выражений для указания правильного порядка выполнения операторов.

Следующий пример вызовет ошибку компиляции:

```
var
  A, B: Integer;
begin
  if A = 0 and B <> 0 then ... // так делать НЕЛЬЗЯ
```

Ошибка связана с тем, что компилятор в первую очередь пытается выполнить побитовый оператор `and` в середине выражения и в результате получается `(0 and B)` - побитная операция, возвращающая целочисленную величину. Далее компилятор выполняет оператор "равно" и получает булевскую величину `A = (0 and B)`. Финальная ошибка оказывается связана с попыткой сравнить булевскую величину `A = (0 and B)` с целочисленной величиной `0`.

Правильно записывать это условие в следующем виде:

```
var
  A, B: Integer;
begin
  if (A = 0) and (B <> 0) then ...
```

Паскаль использует *"короткую оценку (short-circuit evaluation)"* - мощную оптимизацию, позволяющую не вычислять выражение целиком, если какая-либо его часть полностью определяет результат. Рассмотрим пример:

```
if MyFunction(X) and MyOtherFunction(Y) then...
```

- Значение функции `MyFunction(X)` всегда рассчитывается первым.

- И если `MyFunction(X)` вернёт значение `false`, это означает, что мы уже знаем результат всего выражения - какое бы ни было второе значение, при выполнении `false and что-нибудь` мы всегда получим `false`. Таким образом `MyOtherFunction(Y)` вообще не будет выполняться.
- Идентичная ситуация и с выражением `or`. В данном случае, если мы наперёд знаем, что результат будет `true` потому что первый аргумент имеет значение `true`, второй аргумент не влияет на результат и не будет рассчитываться.
- Это особенно полезно если нужно записать выражение типа:

```
if (A <> nil) and A.IsValid then...
```

  В данном случае не возникнет ошибки даже в случае если `A` имеет значение `nil` (т.е. нулевой указатель).

2.5. Проверка одного выражения на множественные значения (оператор case)

Если в зависимости от разных значений определённого выражения должны быть выполнены разные действия, тогда может оказаться удобной конструкция `case .. of .. end`.

```
case SomeValue of
  0: DoSomething;
  1: DoSomethingElse;
  2: begin
       IfItsTwoThenDoThis;
       AndAlsoDoThis;
     end;
  3..10: DoSomethingInCaseItsInThisRange;
  11, 21, 31: AndDoSomethingForTheseSpecialValues;
  else DoSomethingInCaseOfUnexpectedValue;
end;
```

Условие `else` опционально (и соответствует `default` в C-подобных языках). В случае, если текущее значение анализируемого выражения не совпадает ни с одним из описанных случаев и нет условия `else`, то программа просто пропустит всю конструкцию `case` и будет выполняться далее.

Программисты C-подобных языков могут сравнить `case` с весьма подобной конструкцией `switch` в этих языках. Стоит отметить, что `case` в

Паскале защищён от случайного выполнения следующей инструкции, т.е. нет необходимости уделять внимание тому, чтобы размещать инструкцию `break` в конце каждого блока. После выполнения ветви условия программа автоматически закончит обработку конструкции `case` и продолжит работу далее.

2.6. Перечисляемый и порядковый типы, наборы и массивы постоянной длины

Перечисляемый тип (enumerated) в Паскале является очень удобным и прозрачным. Возможно, Вам он понравится и Вы будете использовать его чаще чем перечисляемые типы в других языках :)

```
type
  TAnimalKind = (akDuck, akCat, akDog);
```

Общепринято, что префикс перечисляемого типа состоит из двух букв сокращения имени типа, следовательно `ak` = сокращение для *"Animal Kind"*. Это полезное соглашение, так как имена перечисляемых типов находятся в глобальном пространстве переменных unit-а. Таким образом, с помощью префикса `ak` автоматически уменьшаются шансы на случайный конфликт с другими идентификаторами.

ПРИМЕЧАНИЕ: Конфликты в именах не приводят к неработоспособности программы. Вполне допустимо в различных unit-ах определять одинаковые идентификаторы. Однако, желательно избегать подобных конфликтов везде, где это возможно, чтобы код был прост для понимания и анализа.

ПРИМЕЧАНИЕ: Можно избежать попадания имён перечисляемых типов в глобальное пространство с помощью директивы компилятора `{$scopedenums on}`. В таком варианте будет необходимо обращаться к ним через имя типа следующим образом: `TAnimalKind.akDuck`. В результате отпадает необходимость в префиксе `ak`, и можно просто оставить исходные названия `Duck, Cat, Dog`. Такое исполнение подобно тому, как работают перечисляемые списки в C#.

Прозрачность перечисляемого типа означает, что он не совместим напрямую с целочисленными величинами. Тем не менее, если такая совместимость необходима, можно использовать `Ord(MyAnimalKind)`, чтобы вручную привести список к целочисленному типу. Обратная операция будет выглядеть

как приведение типа `TAnimalKind(MyInteger)` и превратит целое число в соответствующий перечисляемый тип. В последнем случае необходимо также быть уверенным, что `MyInteger` является частью диапазона от `0` до `Ord(High(TAnimalKind))`.

Перечисляемые типы могут быть также использованы в качестве индексов массивов фиксированной длины:

```
type
  TArrayOfTenStrings = array [0..9] of string;
  TArrayOfTenStrings1Based = array [1..10] of string;

  TMyNumber = 0..9;
  TAlsoArrayOfTenStrings = array [TMyNumber] of string;

  TAnimalKind = (akDuck, akCat, akDog);
  TAnimalNames = array [TAnimalKind] of string;
```

Они также могут использоваться для создания наборов (они же set-ы, они же внутренние битовые поля):

```
type
  TAnimalKind = (akDuck, akCat, akDog);
  TAnimals = set of TAnimalKind;
var
  A: TAnimals;
begin
  A := [];
  A := [akDuck, akCat];
  A := A + [akDog];
  A := A * [akCat, akDog];
  Include(A, akDuck);
  Exclude(A, akDuck);
end;
```

2.7. Циклы (for, while, repeat, for .. in)

```
{$mode objfpc}{$H+}{$J-}
{$R+} // включаем проверку на диапазон величин, очень полезно для отладки
var
  MyArray: array [0..9] of Integer;
  I: Integer;
```

```
begin
  // инициализация
  for I := 0 to 9 do
    MyArray[I] := I * I;

  // отображение
  for I := 0 to 9 do
    WriteLn('Квадрат составляет ', MyArray[I]);

  // делает то же самое, что и предыдущий вариант
  for I := Low(MyArray) to High(MyArray) do
    WriteLn('Квадрат составляет ', MyArray[I]);

  // делает то же самое
  I := 0;
  while I < 10 do
  begin
    WriteLn('Квадрат составляет ', MyArray[I]);
    I := I + 1; // это идентично "I += 1" или "Inc(I)"
  end;

  // делает то же самое
  I := 0;
  repeat
    WriteLn('Квадрат составляет ', MyArray[I]);
    Inc(I);
  until I = 10;

  // делает то же самое
  // обратите внимание, тут переменная I перечисляет значения элементов
 массива, а не его индексы
  for I in MyArray do
    WriteLn('Квадрат составляет ', I);
end.
```

Примечания:

- Может показаться, что различие между циклами `while` и `repeat` лишь "косметические" с единственным отличием, что условие записано "с точностью до наоборот": в случае `while .. do` выполнение *продолжается*, пока условие *истинно*, а при `repeat .. until` - выполнение *прекращается*, когда условие *истинно*. Впрочем, есть ещё одно важное отличие: в случае `repeat`, условие проверяется не *в начале*, а *в конце* цикла. Поэтому содержимое цикла `repeat` всегда выполняется как минимум один раз.

- Конструкция `for I := .. to .. do …` похожа на C-подобный цикл `for`. Тем не менее, она более ограничена, поскольку невозможно указать произвольное действие/условие чтобы контролировать итерации цикла. В Паскале `for` используется строго для итерации через последовательные числа (или другие порядковые типы). Единственной уступкой является возможность использования `downto` вместо `to`, чтобы производить счёт в обратном порядке.

  С другой стороны, это существенно облегчает понимание кода, и лучше для оптимизации в его исполнении. Например, *значения верхней и нижней границы вычисляются лишь один раз*, до начала исполнения цикла.

  Следует также обратить внимание, что переменная, которая использовалась для цикла (в примере выше - `I`) становится неопределённой после окончания цикла кроме случая досрочного выхода из цикла с помощью команд `Break` или `Exit`. Цикл `for I in .. do ..` такой же как конструкция foreach в многих других языках и хорошо понимает организацию всех встроенных типов:

- Он может перебирать все значения массива (см. пример выше).
- Он может перебирать все возможные значения перечисляемого типа:

```
var
  AK: TAnimalKind;
begin
  for AK in TAnimalKind do...
```

  - Он может перебирать все элементы набора:

```
var
  Animals: TAnimals;
  AK: TAnimalKind;
begin
  Animals := [akDog, akCat];
  for AK in Animals do ...
```

  - И так же работает на всех пользовательских типах, включая generic-и, например, `TObjectList` или `TFPGObjectList`.

```
{$mode objfpc}{$H+}{$J-}
uses
  SysUtils, FGL;
```

```
type
  TMyClass = class
    I, Square: Integer;
  end;
  TMyClassList = specialize TFPGObjectList<TMyClass>;

var
  List: TMyClassList;
  C: TMyClass;
  I: Integer;
begin
  List := TMyClassList.Create(true); // значение true означает, что
 List владеет всеми дочерними объектами
  try
    for I := 0 to 9 do
    begin
      C := TMyClass.Create;
      C.I := I;
      C.Square := I * I;
      List.Add(C);
    end;

    for C in List do
      WriteLn('Квадрат ', C.I, ' составляет ', C.Square);
  finally
    FreeAndNil(List);
  end;
end.
```

Мы ещё не рассматривали концепцию классов, поэтому последний пример может показаться неочевидным. Но мы обязательно рассмотрим этот вопрос чуть позже :)

2.8. Вывод информации и логов

Для простого вывода строки в Паскале используется процедура `Write` или `WriteLn`. Во втором случае в конце автоматически добавляется символ переноса строки.

Это "волшебная" процедура в Паскале, Она принимает переменное число аргументов, которые могут иметь практически любой тип. Они все будут автоматически сконвертированы в строчный тип при выводе. Кроме того, можно

добавить специальный синтаксис для указания каким образом отформатировать число.

```
WriteLn('Hello world!');
WriteLn('Можно вывести целое число: ', 3 * 4);
WriteLn('Отформатировать его: ', 666:10);
WriteLn('А также вывести число с плавающей запятой: ', Pi:1:4);
```

Чтобы явным образом добавить перенос строки можно использовать константу `LineEnding` (из библиотеки FPC RTL). (*Castle Game Engine* имеет также более краткий вариант `NL`). В отличии от HTML и других подобных синтаксисов разметок в Паскале обратная косая (`\`) не позволяет вставлять специальные символы в строке, например:

```
WriteLn('Первая строка.\nВторая стока.'); // НЕВЕРНЫЙ пример
```

не вставит перенос строки, а просто выдаст все символы этой строки в одной строчке. Правильно делать следующим образом:

```
WriteLn('Первая строка.' + LineEnding + 'Вторая строка.');
```

или так:

```
WriteLn('Первая строка.');
WriteLn('Вторая строка.');
```

Стоит отметить, что функции `write`/`WriteLn` будут работать только в *консольных* приложениях. Для этого необходимо указывать `{$apptype CONSOLE}` (**но не** `{$apptype GUI}`) в главном файле программы. На некоторых ОС консоль явно или скрыто присутствует всегда (Unix) и эта директива не используется. А в некоторых системах попытка выполнения `write`/`WriteLn` из GUI приложения приведёт к ошибке (например, в Windows).

В **Castle Game Engine** не советуется использовать `WriteLn`, поскольку для этого есть специальная функция `WriteLnLog` или `WriteLnWarning` для вывода логов и отладочной информации. Их результат всегда будет направлен в полезном направлении: для Unix-подобных систем это будет стандартный вывод в консоль. Для Windows GUI приложений это будет лог-файл. В Android вывод будет направлен в *Android logging facility* (инструмент логов Андроида),

который можно просматривать с помощью команды `adb logcat`. Использовать `WriteLn` есть смысл лишь в ограниченном наборе случаев, например, для консольных приложений (исполняемых из командной строки) в которых можно быть точно уверенным, что *стандартный вывод* определён. Например, так можно делать в конвертере или генераторе трёхмерных моделей, который предназначен для запуска из командной строки.

2.9. Преобразование данных в строчный тип

Для преобразования произвольного количества аргументов в строку (вместо того, чтобы напрямую выводить их в терминал консоли) есть несколько возможных подходов.

- Некоторые конкретные типы можно преобразовать в строку используя специальные функции, такие как `IntToStr` и `FloatToStr`. В дальнейшем суммировать (concatenation) строки в Паскале можно просто используя знак сложения. Таким образом можно создавать составные строки: `'Моё целое число ' + IntToStr(MyInt) + ', а значение числа пи составляет ' + FloatToStr(Pi)`.
  - *Преимущество*: это очень удобно. Существует множество готовых функций типа `XxxToStr` и им подобных (например, `FormatFloat`), для множества различных типов данных.
  - *Второе преимущество*: они почти всегда имеют обратную функцию. Чтобы преобразовать строку (например, введённую пользователем) в целое или дробное число можно использовать `StrToInt`, `StrToFloat` и подобные (например, `StrToIntDef`).
  - *Недостаток*: длинная сумма множества `XxxToStr` и строк выглядит некрасиво.
- Функция `Format` используется в виде `Format('%d %f %s', [MyInt, MyFloat, MyString])`. Она подобна функции `sprintf` в C-подобных языках. Она вставляет аргументы в соответствующие placeholder-ы согласно заданному образцу. Эти placeholder-ы могут использовать специальный синтаксис, влияющий на форматирование, например `%.4f` это дробный формат с 4 знаками после запятой.
  - *Преимущество*: отделение *строки* от *аргументов* выглядит чисто и красиво. Легко изменить текст строки, не изменяя аргументов, например, если необходимо выполнить перевод.

  - *Второе преимущество*: не используются "волшебные" свойства компилятора. Можно использовать идентичный синтаксис для передачи произвольного количества аргументов произвольного типа в пользовательских процедурах (для этого определите принимаемый параметр как `array of const`). Затем можно передать эти аргументы функции `Format`, или разобрать на список параметров и делать с ним всё, что угодно.
  - *Недостаток*: компилятор не проверяет, соответствует ли строка-образец аргументам. Используя неверный тип placeholder-а приведёт к ошибке `EConvertError`, впрочем, гораздо более понятной, чем segmentation fault (наиболее часто SIGSEGV).

- В Паскале также существует функция `WriteStr(TargetString, …)` во многом подобна базовой функции `Write(…)`, с одним отличием - результат сохраняется в `TargetString`.

  - *Преимущество*: эта функция имеет все возможности функции `Write`, в том числе и специальный "волшебный" синтаксис для форматирования, как например `Pi:1:4`.
  - *Недостаток*: такой специальный синтаксис для форматирования является "волшебным", т.е. он написан специально для конкретной процедуры. Часто это приводит к проблемам, например, невозможно создать свою функцию `MyStringFormatter(…)` которая бы принимала такой синтаксис, как `Pi:1:4`. Именно по этому, а также из-за того, что эта функция долгое время не была доступна в основных компиляторах, такая конструкция не очень популярна.

3. Модули (Unit-ы)

Unit-ы позволяют группировать общие функции и объекты (любые элементы языка, которые могут быть объявлены), для использования другими unit-ами и программами. Они эквивалентны *модулям* и *пакетам* в других языках. Они имеют секцию `interface`, где объявляются доступные для других unit-ов и программ переменные, функции и т.п., и секцию `implementation`, где описано, как они работают. Unit `MyUnit` можно сохранить под именем `myunit.pas` (название должно состоять из строчных букв с расширением `.pas`).

```
{$mode objfpc}{$H+}{$J-}
unit MyUnit;
```

```
interface

procedure MyProcedure(const A: Integer);
function MyFunction(const S: string): string;

implementation

procedure MyProcedure(const A: Integer);
begin
  WriteLn('A + 10 составляет: ', A + 10);
end;

function MyFunction(const S: string): string;
begin
  Result := S + 'строки управляются автоматически';
end;

end.
```

Файл основной программы чаще всего сохраняется в виде файлов типа `myprogram.lpr` (`lpr` = Lazarus program file; в Delphi используются `.dpr`). Следует отметить, что возможны и другие расширения, например, некоторые проекты используют расширение `.pas` для основного файла программы. Для unit-ов изредка используются расширение `.pp`. Лично же я предпочитаю использовать стандартные `.pas` для unit-ов и `.lpr` для FPC/Lazarus программ.

Программа может подключать unit с помощью ключевого слова `uses`:

```
{$mode objfpc}{$H+}{$J-}

program MyProgram;

uses MyUnit;

begin
  WriteLn(MyFunction('Примечание: '));
  MyProcedure(5);
end.
```

Ещё одной полезной особенностью unit-ов являются под-секции `initialization` и `finalization` внутри секции `implementation` которые выполняются при запуске и завершении выполнения программы.

```
{$mode objfpc}{$H+}{$J-}
unit initialization_finalization;
interface

implementation

initialization
  WriteLn('Hello world!');
finalization
  WriteLn('Goodbye world!');
end.
```

3.1. Перекрёстные ссылки между unit-ами

Не только основная программа, но и unit-ы также могут ссылаться на другие unit-ы. Другой unit может войти в секцию interface или только в implementation. Первый вариант позволяет создавать новые определения (процедуры, типы…), используя или наследуя информацию из другого unit-а. Во втором варианте возможности более ограничены - если использовать unit в секции implementation, то применить его идентификаторы возможно лишь в рамках implementation данного unit-а.

```
{$mode objfpc}{$H+}{$J-}
unit AnotherUnit;
interface

uses
  Classes;

{ Тип класса "TComponent" определён в unit-е Classes.
  По этому необходимо использовать uses Classes, как видно выше. }
procedure DoSomethingWithComponent(var C: TComponent);

implementation

uses
  SysUtils;

procedure DoSomethingWithComponent(var C: TComponent);
begin
  { Процедура FreeAndNil определена в unit-е SysUtils.
    Поскольку мы лишь ссылаемся на её имя в разделе implementation,
    вполне допустимо использовать SysUtils в секции "implementation". }
  FreeAndNil(C);
```

```
end;

end.
```

Запрещено применять *кольцевую взаимозависимость (cyclic reference)* в разделе interface. Т.е. два unit-а не могут использовать друг друга в разделе interface. Причина такого ограничения заключается в том, что для того, чтобы "понять" секцию interface unit-а, компилятор анализирует и "понимает" все unit-ы, перечисленные в uses в секции interface. В Паскале это правило придерживается строго, что позволяет достичь высокой скорости компиляции и полностью автоматическое определение компилятором *что именно необходимо перекомпилировать*. В Паскале нет необходимости создания сложных `Makefile` для выполнения простой задачи компиляции, а также нет нужды *перекомпилировать всё* лишь для того, чтобы удостовериться, что все зависимости правильно обновились.

Вполне возможно создавать кольцевые зависимости между unit-ами когда один из них "используется" только в implementation. Поэтому нормально для `A` использовать `B` в interface, и затем unit `B` использует `A` в implementation.

3.2. Определение идентификаторов с указанием имени unit-а

Различные unit-ы могут определять одинаковые идентификаторы. Чтобы поддерживать код простым для чтения и правки, обычно следует избегать таких совпадений, но не всегда это возможно. В таких случаях, последний unit в списке `uses` "перетягивает одеяло на себя", т.е. идентификаторы определённые в нём скрывают одноимённые идентификаторы введённые другими unit-ами ранее.

Однако, возможно недвусмысленно определить unit предоставляющий идентификатор, с помощью конструкции `MyUnit.MyIdentifier`. Это стандартное решение ситуации, когда используемый идентификатор из `MyUnit` скрыт идентификатором из другого unit-а. Для достижения этой же цели можно просто перестроить порядок unit-ов в списке uses, однако это повлияет на все идентификаторы, что не всегда желательно.

```
{$mode objfpc}{$H+}{$J-}
program showcolor;

// unit-ы Graphics и GoogleMapsEngine определяют свой тип TColor.
```

```
uses Graphics, GoogleMapsEngine;

var
  { Это сработает не так, как ожидается, поскольку TColor
    определяется последним unit-ом в списке - GoogleMapsEngine. }
  // Color: TColor;
  { А так будет правильно. }
  Color: Graphics.TColor;
begin
  Color := clYellow;
  WriteLn(Red(Color), ' ', Green(Color), ' ', Blue(Color));
end.
```

В случае unit-ов следует также помнить, что они могут иметь два списка `uses`: один - в секции `interface`, другой - в `implementation`. Основное правило в этом случае звучит как: "*позднейшие unit-ы скрывают все что было до этого*" и применяется последовательно, что в свою очередь означает, что *unit-ы использованные в секции implementation* могут скрывать идентификаторы из *unit-ов использованных в секции interface*. Кроме того, не стоит забывать, что в процессе обработки секции interface данного unit-а компилятором, влияют лишь unit-ы использованные в секции interface. Это может сбить с толку в ситуациях, когда два на вид одинаковых объявления обрабатываются компилятором по-разному:

```
{$mode objfpc}{$H+}{$J-}
unit UnitUsingColors;

// НЕВЕРНЫЙ пример

interface

uses Graphics;

procedure ShowColor(const Color: TColor);

implementation

uses GoogleMapsEngine;

procedure ShowColor(const Color: TColor);
begin
  // WriteLn(ColorToString(Color));
end;
```

```
end.
```

Unit `Graphics` (из набора библиотек Lazarus LCL) определяет тип `TColor`. Но компилятор указывает на ошибку в этом unit-е, указывая на то, что заявленная в секции Interface процедура `ShowColor` не описана. Проблема в том, что unit `GoogleMapsEngine` также определяет тип `TColor`, который используется только в секции `implementation`, следовательно он *перекрывает* определение `TColor` в секции `implementation`. Т.е. компилятор видит это буквально как:

```
{$mode objfpc}{$H+}{$J-}
unit UnitUsingColors;

// НЕВЕРНЫЙ пример
// демонстрирующий, как предыдущий пример "видит" компилятор

interface

uses Graphics;

procedure ShowColor(const Color: Graphics.TColor);

implementation

uses GoogleMapsEngine;

procedure ShowColor(const Color: GoogleMapsEngine.TColor);
begin
  // WriteLn(ColorToString(Color));
end;

end.
```

В данном случае решение тривиальное: необходимо просто изменить `implementation`, чтобы явно использовать `TColor` из unit-а `Graphics`. Это также можно исправить, переместив `GoogleMapsEngine` в секцию interface до unit-а `Graphics`. Впрочем, это может привести к другим последствиям внутри unit-а `UnitUsingColors`, так как коснётся всех его определений.

```
{$mode objfpc}{$H+}{$J-}
unit UnitUsingColors;
```

```
interface

uses Graphics;

procedure ShowColor(const Color: TColor);

implementation

uses GoogleMapsEngine;

procedure ShowColor(const Color: Graphics.TColor);
begin
  // WriteLn(ColorToString(Color));
end;

end.
```

3.3. Передача идентификаторов одного unit-а через другой

Иногда возникает необходимость взять идентификатор из одного unit-а и передать его через другой unit. Т.е. в результате использование нового unit-а должно сделать доступным старый идентификатор в пространстве имён.

В некоторых случаях это делается из-за необходимости сохранить обратную совместимость с предыдущими версиями unit-а. А иногда таким образом удобно "скрыть" какой-либо unit для внутреннего пользования.

Это может быть осуществлено с помощью повторного объявления идентификатора в новом unit-е.

```
{$mode objfpc}{$H+}{$J-}
unit MyUnit;

interface

uses Graphics;

type
  { Используем TColor из unit-а Graphics для определения TMyColor. }
  TMyColor = TColor;
```

```
  { Как вариант, можно переопределить его под тем же именем.
    В таком варианте необходимо будет явно указать наименование unit-a,
    иначе получится несогласованное определение "TColor = TColor". }
  TColor = Graphics.TColor;

const
  { С константами это тоже работает. }
  clYellow = Graphics.clYellow;
  clBlue = Graphics.clBlue;

implementation

end.
```

Стоит отметить, что данный трюк не пройдёт с глобальными процедурами, функциями и переменными. В таком случае возникнет необходимость объявить постоянный указатель на процедуру в другом unit-е (см. Section 7.2, “Колбэки ~~# #~~они же события, они же указатели на функции, они же процедурные переменные”), но такой код выглядит не совсем чисто.

Более оптимальным решением является создание тривиальной "функции-обёртки", которая под видом простого вызова функции из внешнего unit-а, просто передаёт ему параметры и возвращает принимаемые значения обратно.

Чтобы проделать то же с глобальными параметрами иногда используются глобальные (на уровне unit-а) свойства, см. Section 4.3, “Свойства”.

4. Классы

4.1. Основы

В Паскале для объектно-ориентированного программирования чаще всего используются классы (classes). На базовом уровне класс просто является "контейнером", который может вмещать в себя:

- *поля (field)* (иными словами *"переменная внутри класса"*),
- *методы (method)* (иными словами *"процедура или функция внутри класса"*),
- *свойства (property)* (удобный синтаксис для конструкции подобной полю, однако в действительности являющейся парой методов, используемых для *чтения (getter)* и *записи (setter)* чего-либо; детальнее см. ???).

- Вообще говоря, в классах можно разместить очень много различных вещей, см. Section 8.3, “Дополнительные возможности объявления классов и локальные классы”, но об этом пойдёт речь чуть позже.

```
type
  TMyClass = class
    MyInt: Integer; // это "поле"
    property MyIntProperty: Integer read MyInt write MyInt; // это
 "свойство"
    procedure MyMethod; // это "метод"
  end;

procedure TMyClass.MyMethod;
begin
  WriteLn(MyInt + 10);
end;
```

4.2. Наследование (Inheritance), проверка (is), и приведение типов (as)

Паскаль поддерживает наследование и виртуальные методы объектно-ориентированного программирования.

```
{$mode objfpc}{$H+}{$J-}
program MyProgram;

uses
  SysUtils;

type
  TMyClass = class
    MyInt: Integer;
    procedure MyVirtualMethod; virtual;
  end;

  TMyClassDescendant = class(TMyClass)
    procedure MyVirtualMethod; override;
  end;

procedure TMyClass.MyVirtualMethod;
begin
  WriteLn('TMyClass отображает MyInt + 10: ', MyInt + 10);
```

```
end;

procedure TMyClassDescendant.MyVirtualMethod;
begin
  WriteLn('TMyClassDescendant отображает MyInt + 20: ', MyInt + 20);
end;

var
  C: TMyClass;
begin
  C := TMyClass.Create;
  try
    C.MyVirtualMethod;
  finally
    FreeAndNil(C);
  end;

  C := TMyClassDescendant.Create;
  try
    C.MyVirtualMethod;
  finally
    FreeAndNil(C);
  end;
end.
```

По умолчанию методы не являются виртуальными и для объявления их виртуальными необходимо использовать ключевое слово `virtual`. Перекрытие или замещение виртуального метода осуществляется с помощью ключевого слова `override`, иначе компилятор выдаст ошибку. Чтобы скрыть метод без перекрытия используется ключевое слово `reintroduce`, впрочем, без особых на то причин, так делать не стоит.

Чтобы узнать, является ли некоторый класс из семейства классов конкретным его наследником можно использовать оператор `is`. Для выполнения приведения типа класса к конкретному классу следует использовать оператор `as`.

```
{$mode objfpc}{$H+}{$J-}
program is_as;

uses
  SysUtils;

type
  TMyClass = class
```

```
    procedure MyMethod;
  end;

  TMyClassDescendant = class(TMyClass)
    procedure MyMethodInDescendant;
  end;

procedure TMyClass.MyMethod;
begin
  WriteLn('Это MyMethod')
end;

procedure TMyClassDescendant.MyMethodInDescendant;
begin
  WriteLn('Это MyMethodInDescendant')
end;

var
  Descendant: TMyClassDescendant;
  C: TMyClass;
begin
  Descendant := TMyClassDescendant.Create;
  try
    Descendant.MyMethod;
    Descendant.MyMethodInDescendant;

    { производные классы сохраняют все функции родительского класса
      TMyClass, по этому можно таким образом создавать ссылку }
    C := Descendant;
    C.MyMethod;

    { так не сработает, поскольку в TMyClass не определён этот метод }
    //C.MyMethodInDescendant;
    { правильно записать следующим образом: }
    if C is TMyClassDescendant then
      (C as TMyClassDescendant).MyMethodInDescendant;

  finally
    FreeAndNil(Descendant);
  end;
end.
```

Вместо приведения типа в виде `X as TMyClass`, можно использовать приведение типа *без проверки* с помощью выражения `TMyClass(X)`. Такой код будет работать чуть-чуть быстрее, но может привести к неопределённому

поведению в случае если `X` не является наследником `TMyClass`. По этому конструкцию `TMyClass(X)` лучше не применять кроме тех случаев, когда абсолютно очевидно, что `X` действительно является наследником `TMyClass`, например, если до этого тип класса был проверен с помощью оператора `is`:

```
if A is TMyClass then
  (A as TMyClass).CallSomeMethodOfMyClass;
// вариант ниже - работает незначительно быстрее
if A is TMyClass then
  TMyClass(A).CallSomeMethodOfMyClass;
```

4.3. Свойства

Свойства (Properties) являются *"синтаксическим сахаром"* (*прим. перев. syntax sugar - жаргон, означающий синтаксические возможности, применение которых не влияет на поведение программы, но делает использование языка более удобным для программиста*) который можно использовать с целью:

1. Сделать что-то внешнее похожее на поле (может быть прочитано и установлено), но реализовано вызовом функций *считывания значения (getter)* и *установки значения (setter)*. Самое стандартное применение такого подхода - автоматическое выполнннение дополнительных действий каждый раз, когда некоторое значение изменяется.
2. Сделать что-то внешне похожее на поле, но доступное только для чтения - нечто, похожее на константу или функцию без параметров.

```
type
  TWebPage = class
  private
    FURL: string;
    FColor: TColor;
    function SetColor(const Value: TColor);
  public
    { Значение URL невозможно установить напрямую.
      Следует вызвать метод вроде Load('http://www.freepascal.org/'),
      для загрузки страницы и установки значения этого свойства. }
    property URL: string read FURL;
    procedure Load(const AnURL: string);
    property Color: TColor read FColor write SetColor;
  end;
```

```
procedure TWebPage.Load(const AnURL: string);
begin
  FURL := AnURL;
  NetworkingComponent.LoadWebPage(AnURL);
end;

function TWebPage.SetColor(const Value: TColor);
begin
  if FColor <> Value then
  begin
    FColor := Value;
    { Например, требовать обновления объекта, каждый раз,
      когда изменяется значение его цвета:
    Repaint;
    { Ещё пример: обеспечить чтобы нечто изменялось синхронно
      с установкой цвета, например }
    RenderingComponent.Color := Value;
  end;
end;
```

Стоит обратить внимание, что вместо того, чтобы указать метод для чтения или записи, можно напрямую указать читаемое/записываемое поле (которое обычно является private и весьма часто имеет название идентичное property с дополнительным префиксом f (от field - поле)) чтобы непосредственно получать или устанавливать значение. В примере выше, свойство `Color` использует *setter*-метод `SetColor`. Однако, для получения значения свойство `Color` напрямую ссылается на private поле `FColor`. Прямая ссылка на конкретное поле, очевидно, быстрее, чем написание дополнительных методов getter или setter - как с точки зрения разработки, так и с точки зрения скорости исполнения программы.

При объявлении свойства указывается:

1. Может ли оно быть прочитано, и как (с помощью прямого чтения поля, или с использованием метода `getter`).
2. Может ли оно быть установлено, и как (с помощью прямой записи поля, или вызовом метода `setter`).
3. Имеет ли оно значение по умолчанию и какое (с помощью указания значения по умолчанию после ключевого слова `default`).

Компилятор следит за тем, чтобы типы и параметры соответствующих полей и методов совпадали с типом свойства с которым они работают. Например, чтобы

прочитать свойство `Integer` следует или предоставить поле `Integer`, или беспараметрический метод (функцию), который возвращает `Integer`.

С технической точки зрения, методы "getter" и "setter" - обычные методы и они могут делать абсолютно что угодно (включая массу дополнительных функций). Всё же правилом хорошего тона является создание таких свойств, которые ведут себя более-менее подобно обычному полю:

- Функция *getter* не должна иметь невидимых побочных эффектов (например, она не должна читать некоторый ввод из файла / с клавиатуры). Её значение должно быть детерминистическим (без рандомизации или псевдо-рандомизации :)) - чтение свойства должно всегда иметь смысл и возвращать одинаковый результат, если между операциями чтения ничего не изменилось.

  Следует отметить, что вполне нормально если выполнение *getter* имеет некие *невидимые* последствия, например, сохранение в кеше результатов какого-либо вычисления для ускорения выполнения кода при следующем вызове. По факту, это одна из очень полезных возможностей функции "getter".

- Функция *setter* должна всегда устанавливать значение таким образом, чтобы *getter* вернул его же обратно. Не стоит автоматически отбрасывать неверные значения "setter", а в случае, когда это необходимо, следует вызвать exception. Также не желательно конвертировать или масштабировать запрашиваемое значение. Главная идея заключается в том, чтобы после установки `MyClass.MyProperty := 123;` можно было с уверенностью сказать, что `MyClass.MyProperty = 123`.

- *Свойства, доступные только для чтения* часто используют для создания неких полей доступных из внешнего кода только для чтения. Снова таки, хорошая практика - делать их поведение похожим на константу, по крайней мере для данного экземпляра объекта в его текущем состоянии. Значение такого свойства не должно меняться неожиданно. *Если необходимо возвращать что-то случайное, лучше сделать функцию, а не свойство.*

- Поле, к которому обращаются свойства почти всегда находится в разделе *private*, поскольку главная идея свойств - служить обёрткой и методом доступа к нему.

- Технически, возможно создать свойства, которые только устанавливают значение, но не читают его. Впрочем, хороших примеров такой реализации лично мне ещё не встречалось :)

ПРИМЕЧАНИЕ: Свойства так же могут быть определены вне класса, на уровне unit-а. Они служат аналогичной цели: они внешне выглядят как глобальные переменные, но доступ к ним вызывает соответствующие функции *getter* и *setter*.

4.4. Исключения

В паскале можно вызывать и использовать исключения. Их можно "ловить" с помощью конструкции `try … except … end`, также можно применять секцию "выполнить в конце" `try … finally … end`.

```
{$mode objfpc}{$H+}{$J-}

program MyProgram;

uses
  SysUtils;

type
  TMyClass = class
    procedure MyMethod;
  end;

procedure TMyClass.MyMethod;
begin
  if Random > 0.5 then
    raise Exception.Create('Вызываем exception!');
end;

var
  C: TMyClass;
begin
  Randomize;
  C := TMyClass.Create;
  try
    C.MyMethod;
  finally
    FreeAndNil(C)
  end;
end.
```

Обратите внимание, что раздел `finally` будет выполнен даже в случае, если выполнение будет прекращено командой `Exit` (из функции, процедуры или метода), операторами `Break` или `Continue` (внутри тела цикла).

4.5. Уровни видимости

Как и в большинстве других объектно-ориентированных языков, в Паскале имеются визуальные спецификаторы для ограничения "видимости" полей / методов / свойств.

Основные уровни видимости являются следующими:

`public`
: предоставлен доступ из любого участка кода, включая код в других unit-ах.

`private`
: доступен только в этом классе.

`protected`
: доступен только в этом классе и всех его наследниках.

Краткое описание `private` и `protected`, данное выше, не полностью верно. Код в *текущем unit-е* может преодолевать эти границы, и получать доступ к секции `private` и `protected`. Иногда это полезная особенность, позволяющая реализовывать тесно связанные классы. В остальных же случаях следует использовать `strict private` или `strict protected` для организации полной недоступности данных методов, полей или свойств извне класса. Детальнее этот вопрос рассматривается в разделе Section 8.1, “Различие private и strict private”.

По умолчанию, если видимость не указана явно, то она соответствует `public`. исключение составляют классы, которые объявляются при включённой директиве `{$M+}`, либо наследники классов, которые были скомпилированы при `{$M+}`, что включает в себя всех потомков `TPersistent`, включая потомков `TComponent`, который сам является потомком `TPersistent`. Для таких классов по умолчанию видимость принимается `published`, которая подобна `public`, однако позволяет работать с ними с помощью потоковой (stream) системы.

Однако, не каждому типу поля или свойства позволено быть в секции published - не каждый тип может быть конвертирован в поток (stream), и лишь классы, состоящие из простых полей, могут передаваться потоком. Если нет необходимости создавать потоки, но нужно просто сделать что-то доступное для всех пользователей, то следует использовать `public`.

4.6. Предок по умолчанию

Если явно не объявить родительский класс, то по умолчанию каждый `class` наследует `TObject`.

5. Освобождение памяти классов

5.1. Всегда необходимо освобождать экземпляры класса

Чтобы избежать утечек памяти, все экземпляры класса должны быть освобождены вручную. Хорошей практикой является использование опции компилятора FPC -gl -gh, чтобы обнаруживать утечки памяти (более подробно см. https://castle-engine.io/manual_optimization.php#section_memory).

Следует обратить внимание, что это не касается вызванных исключений (raised exceptions). Не смотря на то, что при вызове исключения действительно создаётся класс (и это вполне обычный класс, для этих целей также можно создавать и свои классы), такой экземпляр класса освобождается автоматически.

5.2. Каким образом освобождать память

Самым удобным методом освобождения класса является операция `FreeAndNil(A)` из unit-а `SysUtils` вызванная для данного экземпляра класса. Она проверяет, не имеет ли `A` значение `nil`, и если нет — вызывает его деструктор (destructor), и устанавливает значение `A` в `nil`. Таким образом повторный вызов данной процедуры не приведёт к ошибке.

Приблизительно это соответствует следующему:

```
if A <> nil then
begin
  A.Destroy;
  A := nil;
end;
```

Впрочем, эта аналогия немного упрощена, поскольку процедура `FreeAndNil` совершает ещё одно полезное действие, сразу устанавливая `A` значение `nil` **до того** как будет вызван destructor данного класса. Это позволяет избежать целой группы ошибок благодаря тому, что "внешний" код не сможет случайно получить доступ к не до конца уничтоженному экземпляру класса.

Иногда можно заметить, что применяется метод `A.Free` который соответствует следующему коду:

```
if A <> nil then
  A.Destroy;
```

Т.е. освобождает класс `A`, если он не равен `nil`.

Стоит отметить, что в нормальных условиях никогда не стоит вызывать метод класса, ссылка на который может оказаться `nil`. По этому `A.Free` может выглядеть подозрительно на первый взгляд, поскольку `A` вполне может иметь значение `nil`. Однако, метод `Free` является исключением из этого правила. Это выглядит немного "грязновато" — а именно, выполняется проверка `Self <> nil`. Такой фокус работает только для не-виртуальных методов (т.е. в случае, если не вызываются виртуальные методы и не требуется доступ к полям класса).

По этому лучше всегда использовать `FreeAndNil(A)`, без исключений, и никогда не использовать метод `Free` или напрямую деструктор `Destroy`. Такой концепции придерживается *Castle Game Engine*. Это позволяет быть уверенным, что все ссылки *либо равны `nil`, либо указывают на существующий рабочий экземпляр класса*.

5.3. Ручное и автоматическое освобождение памяти

Во многих ситуациях, освобождение экземпляра класса не является чем-то сложным. Просто пишется destructor, как пара соответствующему constructor-у, и освобождает все объекты, память для которых была выделена в constructor-е (а, точнее, в продолжении всего времени существования класса). Важно следить за тем, чтобы освобождать каждый объект лишь **один раз**. Хорошей практикой будет всегда присваивать освобождённой ссылке значение `nil`, а наиболее удобно сделать это, вызвав команду `FreeAndNil(A)`.

Например:

```
uses SysUtils;

type
  TGun = class
  end;
```

```
  TPlayer = class
    Gun1, Gun2: TGun;
    constructor Create;
    destructor Destroy; override;
  end;

constructor TPlayer.Create;
begin
  inherited;
  Gun1 := TGun.Create;
  Gun2 := TGun.Create;
end;

destructor TPlayer.Destroy;
begin
  FreeAndNil(Gun1);
  FreeAndNil(Gun2);
  inherited;
end;
```

Чтобы избежать необходимости каждый раз явным образом освобождать экземпляр класса, можно использовать полезную особенность `TComponent`, которая называется *"owner"* (владение дочерним объектом). Объект у которого есть *owner* (владелец) будет автоматически освобождён его *owner-ом*. Механизм очень гибкий и никогда не освобождает объекты, которые уже освобождены, т.е. всё будет работать правильно и для несозданных объектов, и для освобождённых ранее. Таким образом предыдущий пример можно переписать следующим образом:

```
uses SysUtils, Classes;

type
  TGun = class(TComponent)
  end;

  TPlayer = class(TComponent)
    Gun1, Gun2: TGun;
    constructor Create(AOwner: TComponent); override;
  end;

constructor TPlayer.Create(AOwner: TComponent);
begin
  inherited;
```

```
  Gun1 := TGun.Create(Self);
  Gun2 := TGun.Create(Self);
end;
```

Следует обратить внимание, что также необходимо override виртуальный constructor от `TComponent`. Это, в свою очередь, означает, что нельзя изменять параметры constructor-а. Впрочем, всё-таки это возможно — объявив новый constructor с ключевым словом `reintroduce`. Однако здесь стоит быть осторожным, так как некоторый функционал, например, streaming, всё равно будет использовать виртуальный constructor, по этому следует удостовериться, что во всех возможных случаях всё будет работать корректно.

Ещё один удобный механизм автоматического освобождения памяти — функционал `OwnsObjects` (который по умолчанию равен `true`) классов-списков, таких, как `TFPGObjectList` или `TObjectList`. Т.е. можно сделать следующим образом:

```
uses SysUtils, Classes, FGL;

type
  TGun = class
  end;

  TGunList = specialize TFPGObjectList<TGun>;

  TPlayer = class
    Guns: TGunList;
    Gun1, Gun2: TGun;
    constructor Create;
    destructor Destroy; override;
  end;

constructor TPlayer.Create;
begin
  inherited;
  // Вообще говоря, параметр OwnsObjects и так true по умолчанию
  Guns := TGunList.Create(true);
  Gun1 := TGun.Create;
  Guns.Add(Gun1);
  Gun2 := TGun.Create;
  Guns.Add(Gun2);
end;
```

```
destructor TPlayer.Destroy;
begin
  { Здесь достаточно освободить сам список.
    Он сам автоматически освободит всё содержимое. }
  FreeAndNil(Guns);

  { Таким образом нет нужды освобождать Gun1, Gun2 отдельно. Правда,
 хорошей
    практикой будет теперь установить значение "nil" соответствующим
 значениям
    ссылок на них, поскольку мы знаем, что они освобождены.
    В этом простом классе с простым destructor-ом, очевидно,
    что к ним не произойдёт доступа, однако в случае сложных destructor-ов
    это может оказаться полезно.

    Альтернативно, можно избежать объявления Gun1 и Gun2 отдельно
    и использовать напрямую Guns[0] и Guns[1] в коде.
    Можно также создать метод Gun1, который возвращает ссылку на
 Guns[0]. }
  Gun1 := nil;
  Gun2 := nil;
  inherited;
end;
```

Заметим, что механизм owner-ов классов-списков простой (без дополнительных проверок) и, в случае освобождения содержащегося в списке экземпляра класса сторонним кодом, впоследствии возникнет ошибка. Чтобы исключить что-либо из списка без освобождения используется метод `Extract`, однако это также означает, что в дальнейшем элемент придётся освободить вручную.

Например, в **Castle Game Engine** все наследники класса `TX3DNode` автоматически управляют памятью при добавлении другой `TX3DNode` в список children. Корневая X3DNode называется `TX3DRootNode` и, в свою очередь, обычно имеет своим owner-ом класс `TCastleSceneCore`. Другие объекты также имеют простой механизм owner-а — обычно это обозначено параметром или свойством под названием вида `OwnsXxx`.

5.4. Free notification

Если на экземпляр класса создано несколько ссылок, это эквивалентно тому, что две ссылки указывают на одну и ту же область памяти. Если освободить одну из них, вторая окажется *"болтающимся" pointer-ом*. Нельзя пытаться получить доступ к области памяти, которая была освобождена. Это может

привести к runtime ошибке, либо может быть получено неопределённое значение ("мусор") — в случае, если эта область памяти уже была повторно выделена для других элементов внутри текущей программы.

В таком случае не достаточно просто вызывать `FreeAndNil` поскольку эта функция установит `nil` лишь для переданной ей ссылки — автоматического метода для подобных задач не существует. Рассмотрим следующий пример:

```
var
  Obj1, Obj2: TObject;
begin
  Obj1 := TObject.Create;
  Obj2 := Obj1;
  FreeAndNil(Obj1);

  // что произойдёт, если попытаться получить доступ к объекту Obj1 или
 Obj2?
end;
```

1. В конце данного блока ссылка `Obj1` является `nil`. Если необходимо получить доступ к ней в коде программы, для надёжности следует использовать проверку `if Obj1 <> nil then …` чтобы случайно не вызвать метод уже освобождённого экземпляра класса, например:

   ```
   if Obj1 <> nil then
     WriteLn(Obj1.ClassName);
   ```

   Попытка доступа к ссылке `nil` на экземпляр класса приведёт к предсказуемой и понятной ошибке. Таким образом, даже если код не будет проверять `Obj1 <> nil`, и попытается вслепую получить доступ к `Obj1`, возникнет достаточно ясное сообщение об ошибке.

   То же самое происходит и при попытке вызова виртуального, или не-виртуального метода который пытается получить доступ к полю освобождённого экземпляра класса.
2. Ситуация с `Obj2` — куда сложнее. Её значение не `nil`, однако оно уже ошибочно. Попытка доступа к не-`nil` ссылки на несуществующий экземпляр класса приводит к непредсказуемому результату — это может быть и ошибка access violation, а может и просто какое-то случайное значение.

К решению такой проблемы есть несколько путей:

- Первое решение - внимательно читать документацию к объекту. Не предполагать ничего о длительности жизни ссылки, если она создана чужим кодом. Если в классе `TCar` есть поле `wheel`, указывающее на экземпляр класса типа `TWheel`, то есть *правило* что ссылка на `wheel` верна, пока существует класс `car`, и сам `car` освободит все его `wheel` используя свой destructor. Но это правило не всегда возможно выполнить. В более сложных случаях, в документации следует сделать упоминание о том, что и как происходит со ссылками.

- В примере выше, сразу после освобождения экземпляра класса `Obj1`, можно просто вручную установить `Obj2` значение `nil`. В данном конкретном примере тривиально.

- Однако, наиболее перспективным решением будет применение специального механизма класса `TComponent` под названием "free notification" (извещение об освобождении). Таким образом один компонент может получить извещение в случае освобождения одной из компонент, и далее установить ссылку на неё в `nil`.

  Таким образом можно получить *слабую ссылку*. Использовать эту механику можно в различных задачах, например, позволить коду извне изменять ссылки, в том числе, возможность освобождать память в любой момент.

  Для этого оба класса должны наследовать `TComponent`. Обычно это сводится к использованию `FreeNotification` , `RemoveFreeNotification`, и override `Notification`.

  Следующий пример демонстрирует как использовать этот подход вместе с constructor-ом / destructor-ом и setter-ом. Иногда можно всё сделать намного проще, но здесь демонстрируется полномасштабная версия, которая будет верной в любом случае.

```
type
  TControl = class(TComponent)
  end;

  TContainer = class(TComponent)
  private
    FSomeSpecialControl: TControl;
    procedure SetSomeSpecialControl(const Value: TControl);
  protected
```

```
    procedure Notification(AComponent: TComponent; Operation:
 TOperation); override;
  public
    destructor Destroy; override;
    property SomeSpecialControl: TControl
      read FSomeSpecialControl write SetSomeSpecialControl;
  end;

implementation

procedure TContainer.Notification(AComponent: TComponent; Operation:
 TOperation);
begin
  inherited;
  if (Operation = opRemove) and (AComponent = FSomeSpecialControl) then
    { установить значение nil для SetSomeSpecialControl чтобы всё
 аккуратно подчистить }
    SomeSpecialControl := nil;
end;

procedure TContainer.SetSomeSpecialControl(const Value: TControl);
begin
  if FSomeSpecialControl <> Value then
  begin
    if FSomeSpecialControl <> nil then
      FSomeSpecialControl.RemoveFreeNotification(Self);
    FSomeSpecialControl := Value;
    if FSomeSpecialControl <> nil then
      FSomeSpecialControl.FreeNotification(Self);
  end;
end;

destructor TContainer.Destroy;
begin
  { Установить значение nil для SetSomeSpecialControl, чтобы запустить
 notification про освобождение памяти }
  SomeSpecialControl := nil;
  inherited;
end;
```

6. Run-time library

6.1. Ввод/вывод с помощью потоков

Для ввода / выода в современных программах на Паскале используется класс `TStream`. У него также есть множество полезных производных классов, таких как `TFileStream`, `TMemoryStream`, `TStringStream` и т.п.

```
{$mode objfpc}{$H+}{$J-}
uses
  SysUtils, Classes;

var
  S: TStream;
  InputInt, OutputInt: Integer;
begin
  InputInt := 666;

  S := TFileStream.Create('my_binary_file.data', fmCreate);
  try
    S.WriteBuffer(InputInt, SizeOf(InputInt));
  finally
    FreeAndNil(S);
  end;

  S := TFileStream.Create('my_binary_file.data', fmOpenRead);
  try
    S.ReadBuffer(OutputInt, SizeOf(OutputInt));
  finally
    FreeAndNil(S);
  end;

  WriteLn('Из файла прочитано целое число: ', OutputInt);
end.
```

В **Castle Game Engine** следует использовать метод `Download` для создания потока, который оперирует ресурсами (это могут быть файлы, данные, скачанные с URL или Android assets). Более того, для однозначного кросс-платформенного указания на папку игровых данных (обычно это под-папка `data`) следует использовать функцию `ApplicationData`.

```
EnableNetwork := true;
```

```
S := Download('https://castle-engine.io/latest.zip');
```

```
S := Download('file:///home/michalis/my_binary_file.data');
```

```
S := Download(ApplicationData('gui/my_image.png'));
```

Чтобы прочитать обычный текстовый файл лучше использовать класс `TTextReader` из `CastleClassUtils`. Он предоставляет построчный API, как надстройку над `TStream`. URL можно задать через конструктор класса `TTextReader`, либо можно передать ему готовый `TStream` вручную.

```
Text := TTextReader.Create(ApplicationData('my_data.txt'));
while not Text.Eof do
  WriteLnLog('NextLine', Text.ReadLine);
```

6.2. Списки

Для создания различных списков переменной длинны, лично я советую использовать generic классы из unit-а `FGL`. Можно использовать `TFPGList` для простых типов данных (включая record-ы или устаревшие object-ы), и `TFPGObjectList` для списка экземпляров классов. В **Castle Game Engine:** можно также использовать `TGenericStructList` из `CastleGenericLists` для создания списков record-ов или устаревших object-ов. Это позволяет обойти проблему невозможности override их операторов в старых версиях FPC.

Применение таких списков является хорошей идеей по нескольким причинам: они типо-безопасны и их API имеет много полезных функций, таких как поиск, сортировка, итерация и т.п. Вообще говоря, не стоит использовать *динамические массивы* (`array of X`, `SetLength(X, …)`) поскольку их API очень неудобный - можно применить лишь `SetLength`. Также не желательно использовать `TList` или `TObjectList` поскольку это потребует преобразование типа из общего `TObject` в конкретный тип класса при каждом обращении.

6.3. Клонирование классов: TPersistent.Assign

Стандартным подходом для создания копии экземпляра класса является наследование этим классом `TPersistent` с последующим override его метода `Assign`. Здесь нет ничего сложного, просто нужно в методе `Assign` прописать копирование полей, которые Вам необходимы.

Однако, здесь понадобится достаточно аккуратный подход в имплементации метода `Assign`, поскольку копирование может происходить не только из данного класса, а и из производных от него.

```
{$mode objfpc}{$H+}{$J-}
uses
  SysUtils, Classes;

type
  TMyClass = class(TPersistent)
  public
    MyInt: Integer;
    procedure Assign(Source: TPersistent); override;
  end;

  TMyClassDescendant = class(TMyClass)
  public
    MyString: string;
    procedure Assign(Source: TPersistent); override;
  end;

procedure TMyClass.Assign(Source: TPersistent);
var
  SourceMyClass: TMyClass;
begin
  if Source is TMyClass then
  begin
    SourceMyClass := TMyClass(Source);
    MyInt := SourceMyClass.MyInt;
    // Xxx := SourceMyClass.Xxx; // добавить необходимые поля здесь
  end else
    { Вызываем inherited ТОЛЬКО если не получается вручную обработать
 Source }
    inherited Assign(Source);
end;

procedure TMyClassDescendant.Assign(Source: TPersistent);
var
  SourceMyClassDescendant: TMyClassDescendant;
begin
  if Source is TMyClassDescendant then
  begin
    SourceMyClassDescendant := TMyClassDescendant(Source);
    MyString := SourceMyClassDescendant.MyString;
```

```
    // Xxx := SourceMyClassDescendant.Xxx; // добавить необходимые поля
 здесь
  end;

  { ВСЕГДА вызываем inherited, чтобы TMyClass.Assign сама обработала
    все остающиеся поля. }
  inherited Assign(Source);
end;

var
  C1, C2: TMyClass;
  CD1, CD2: TMyClassDescendant;
begin
  // тестируем TMyClass.Assign
  C1 := TMyClass.Create;
  C2 := TMyClass.Create;
  try
    C1.MyInt := 666;
    C2.Assign(C1);
    WriteLn('C2 состояние: ', C2.MyInt);
  finally
    FreeAndNil(C1);
    FreeAndNil(C2);
  end;

  // тестируем TMyClassDescendant.Assign
  CD1 := TMyClassDescendant.Create;
  CD2 := TMyClassDescendant.Create;
  try
    CD1.MyInt := 44;
    CD1.MyString := 'что-нибудь';
    CD2.Assign(CD1);
    WriteLn('CD2 состояние: ', CD2.MyInt, ' ', CD2.MyString);
  finally
    FreeAndNil(CD1);
    FreeAndNil(CD2);
  end;
end.
```

Иногда более удобно написать альтернативный override метода `AssignTo` в копируемом классе, а не делать override метода `Assign` в классе, в который выполняется копирование.

Следует быть осторожным с `inherited` при написании `Assign`. *Inherited* из `TPersistent.Assign` должен вызываться исключительно в случаях, если Вы

не можете самостоятельно выполнить копирование в своём коде (это позволит использовать метод `AssignTo` и создавать сообщения об ошибках, в случае, если копирование не может быть выполнено). С другой стороны, если данный класс является производным от класса, в котором уже есть метод `Assign`, то в данном случае *следует обязательно вызывать inherited* из `TMyClass.Assign`. См. пример выше.

Примечание: Обратите внимание, что наследуя класс `TPersistent` по умолчанию видимость полей становится `published`, чтобы дать возможность переводить в поток (streaming) производные от `TPersistent` классы. Однако, не все типы полей и свойств разрешены в секции `published`. Если возникают связанные с этим ошибки и нет необходимости передавать эти данные в поток, просто вручную измените уровень видимости на `public`. Детальнее см. раздел Section 4.5, “Уровни видимости”.

7. Различные полезные возможности языка

7.1. Местные (вложенные) процедуры

Внутри большой *процедуры* (это может быть функция, процедура, метод и т.п.) можно определить вложенную (местную) под-процедуру.

Таким образом такая локальная под-процедура имеет полный доступ (чтение и запись) всех параметров процедуры, в которой она находится, *если они были объявлены раньше, чем эта процедура*. Это очень удобно, позволяя разделять длинные процедуры на несколько небольших частей без необходимости передавать множество информации в виде параметров. Однако, не следует злоупотреблять этой возможностью. В случае, если местная (вложенная) процедура использует и, тем более, изменяет много переменных процедуры, в которой она находится, такой код становится сложным для понимания.

Данные два примера абсолютно эквивалентны:

```
procedure SumOfSquares(const N: Integer): Integer;

  function Square(const Value: Integer): Integer;
  begin
    Result := Value * Value;
  end;
```

```
var
  I: Integer;
begin
  Result := 0;
  for I := 0 to N do
    Result := Result + Square(I);
end;
```

И второй вариант, в котором `Square` получает прямой доступ к переменной `I`:

```
procedure SumOfSquares(const N: Integer): Integer;
var
  I: Integer;

  function Square: Integer;
  begin
    Result := I * I;
  end;

begin
  Result := 0;
  for I := 0 to N do
    Result := Result + Square;
end;
```

Местные процедуры могут быть многократно вложенными — что означает, что внутри локальной процедуры можно определить местную локальную под-процедуру и т.д. Однако, не желательно этим увлекаться, поскольку в результате код может стать совершенно нечитабельным.

7.2. Колбэки — они же события, они же указатели на функции, они же процедурные переменные

Они позволяют вызвать функцию не прямым указанием её названия, а посредством переменной. Эта переменная может быть назначена во время исполнения кода для указания на любую функцию *с указанными типами параметров и возвращаемых величин*.

Колбэки могут быть:

- Обычными, что означает, что они могут указывать на любую нормальную функцию (т.е. не на метод или локальную функцию)

```
{$mode objfpc}{$H+}{$J-}

function Add(const A, B: Integer): Integer;
begin
  Result := A + B;
end;

function Multiply(const A, B: Integer): Integer;
begin
  Result := A * B;
end;

type
  TMyFunction = function (const A, B: Integer): Integer;

function ProcessTheList(const F: TMyFunction): Integer;
var
  I: Integer;
begin
  Result := 1;
  for I := 2 to 10 do
    Result := F(Result, I);
end;

var
  SomeFunction: TMyFunction;
begin
  SomeFunction := @Add;
  WriteLn('1 + 2 + 3 ... + 10 = ', ProcessTheList(SomeFunction));

  SomeFunction := @Multiply;
  WriteLn('1 * 2 * 3 ... * 10 = ', ProcessTheList(SomeFunction));
end.
```

- Указывающими на метод: для этого в конце добавляется `of object`.

```
{$mode objfpc}{$H+}{$J-}
uses
  SysUtils;

type
  TMyMethod = procedure (const A: Integer) of object;

  TMyClass = class
```

```
    CurrentValue: Integer;
    procedure Add(const A: Integer);
    procedure Multiply(const A: Integer);
    procedure ProcessTheList(const M: TMyMethod);
  end;

procedure TMyClass.Add(const A: Integer);
begin
  CurrentValue := CurrentValue + A;
end;

procedure TMyClass.Multiply(const A: Integer);
begin
  CurrentValue := CurrentValue * A;
end;

procedure TMyClass.ProcessTheList(const M: TMyMethod);
var
  I: Integer;
begin
  CurrentValue := 1;
  for I := 2 to 10 do
    M(I);
end;

var
  C: TMyClass;
begin
  C := TMyClass.Create;
  try
    C.ProcessTheList(@C.Add);
    WriteLn('1 + 2 + 3 ... + 10 = ', C.CurrentValue);

    C.ProcessTheList(@C.Multiply);
    WriteLn('1 * 2 * 3 ... * 10 = ', C.CurrentValue);
  finally
    FreeAndNil(C);
  end;
end.
```

Следует заметить, что *невозможно* передать обычные процедуры или функции как методы. Они несовместимы. Если Вам необходимо использовать колбэк `of object`, но не хотите создавать экземпляр-пустышку для класса, есть возможность передавать Section 8.2, “Class method” в качестве метода.

```
type
  TMyMethod = function (const A, B: Integer): Integer of object;

  TMyClass = class
    class function Add(const A, B: Integer): Integer
    class function Multiply(const A, B: Integer): Integer
  end;

var
  M: TMyMethod;
begin
  M := @TMyClass(nil).Add;
  M := @TMyClass(nil).Multiply;
end;
```

К сожалению, в данном случае приходится писать громоздкую конструкцию `@TMyClass(nil).Add`, а не просто `@TMyClass.Add`.

- Может указывать на локальную процедуру: для этого её нужно объявить с `is nested` в конце, а также установить директиву `{$modeswitch nestedprocvars}` для данного участка кода. Детальнее см. Section 7.1, “Местные (вложенные) процедуры”.

7.3. Generic-и

Generic-и - это очень мощная функция современных языков. Определение чего-либо (обычно - класса) может быть конкретизировано другим типом. Наиболее естественным примером является необходимость создать контейнер (список, словарь, дерево, граф…): можно определить *список типа T*, а потом *specialize* (конкретизировать) его чтобы сразу получить *список из integer*, *список из string*, *список из TMyRecord* и т.д.

Generic-и в Паскале реализованы весьма подобно generic-ам в C++. Это означает, что они "конкретизируются" когда происходит specialize, что *немного* похоже на поведение макросов (однако, когда тип "конкретизирован", используются только эти конкретные определения, и таким образом невозможно "добавить" какое-либо неожиданное поведение в generic после его "конкретизации"). Преимуществом их является высокая скорость выполнения (оптимизированная для каждого конкретного типа) и поддержка типов любого размера. Можно использовать примитивные типы (integer, float) а также record, class и т.п. при specialize данного generic-а.

```
{$mode objfpc}{$H+}{$J-}
uses
  SysUtils;

type
  generic TMyCalculator<T> = class
    Value: T;
    procedure Add(const A: T);
  end;

procedure TMyCalculator.Add(const A: T);
begin
  Value := Value + A;
end;

type
  TMyFloatCalculator = specialize TMyCalculator<Single>;
  TMyStringCalculator = specialize TMyCalculator<string>;

var
  FloatCalc: TMyFloatCalculator;
  StringCalc: TMyStringCalculator;
begin
  FloatCalc := TMyFloatCalculator.Create;
  try
    FloatCalc.Add(3.14);
    FloatCalc.Add(1);
    WriteLn('Сложение величин типа Float: ', FloatCalc.Value:1:2);
  finally
    FreeAndNil(FloatCalc);
  end;

  StringCalc := TMyStringCalculator.Create;
  try
    StringCalc.Add('что-нибудь');
    StringCalc.Add(' ещё');
    WriteLn('Сложение величин типа String: ', StringCalc.Value);
  finally
    FreeAndNil(StringCalc);
  end;
end.
```

Возможности generic-ов не ограничиваются классами, можно создать generic функции и процедуры:

```
{$mode objfpc}{$H+}{$J-}
uses SysUtils;

{ Примечание: этот пример требует FPC 3.1.1 и не скомпилируется в FPC
 3.0.0 или более ранних версиях. }

generic function Min<T>(const A, B: T): T;
begin
  if A < B then
    Result := A
  else
    Result := B;
end;

begin
  WriteLn('Min (1, 0): ', specialize Min<Integer>(1, 0));
  WriteLn('Min (3.14, 5): ', specialize Min<Single>(3.14, 5):1:2);
  WriteLn('Min (''a'', ''b''): ', specialize Min<string>('a', 'b'));
end.
```

7.4. Overload

Методы (а также, глобальные функции и процедуры) с одинаковым именем могут существовать при условии, что они принимают разные параметры. На основании передаваемых процедуре параметров компилятор определяет, какую именно из этих функций использовать в данном случае.

По умолчанию overload использует стиль FPC. Это означает, что абсолютно все имена в пространстве имён (в классе или unit-е) равны, и скрывает все остальные методы в пространстве имён с более низким приоритетом. Например, если создать класс с методом `Foo(Integer)` и `Foo(string)`, и при этом класс наследует класс в котором уже есть `Foo(Float)`, то метод `Foo(Float)` невозможно будет вызвать напрямую, только с помощью приведения типа данного класса к родительскому классу. Чтобы избежать такого поведения, при объявлении процедур и функций следует использовать ключевое слово `overload`.

7.5. Предобработка кода

Можно использовать простые директивы предобработки кода с целью:

- компиляции на основании различных условий (например, если код зависит от платформы, или других заданных вручную параметров),
- чтобы включить один файл внутри другого,
- для вызова беспараметрических макросов.

Обратите внимание, макросы с параметрами запрещены. В общем, следует избегать предобработки кода кроме случаев, когда она действительно необходима. Предобработка происходит перед парсингом, что значит, что она вполне может "ломать" обычный синтаксис языка Паскаль. Это мощно, но немного "грязно".

```
{$mode objfpc}{$H+}{$J-}
unit PreprocessorStuff;
interface

{$ifdef FPC}
{ всё что идёт внутри данного условия ifdef определено только для FPC, а
 не других компиляторов (например, Delphi). }
procedure Foo;
{$endif}

{ Определить константу NewLine. Это пример того, как "нормальный"
  синтаксис Паскаля "поломан" директивами предобработки.
  Если компилировать на Unix-системах (включая Linux, Android, Mac OS X),
  компилятор увидит следующее:

    const NewLine = #10;

  Если компилировать на Windows, компилятор увидит так:

    const NewLine = #13#10;

  Однако, на других операционных системах, код не скомпилируется,
  поскольку компилятор увидит следующее:

    const NewLine = ;

  Вообще, это *хорошо* что в данном случае возникает ошибка -- если
возникнет
  необходимость портировать программу на другую операционную систему,
  которая не является ни Unix, ни Windows, то компилятор "напомнит", что
  необходимо выбрать правильное значение NewLine для такой системы. }
```

```
const
  NewLine =
    {$ifdef UNIX} #10 {$endif}
    {$ifdef MSWINDOWS} #13#10 {$endif} ;

{$define MY_SYMBOL}

{$ifdef MY_SYMBOL}
procedure Bar;
{$endif}

{$define CallingConventionMacro := unknown}
{$ifdef UNIX}
  {$define CallingConventionMacro := cdecl}
{$endif}
{$ifdef MSWINDOWS}
  {$define CallingConventionMacro := stdcall}
{$endif}
procedure RealProcedureName;
 CallingConventionMacro; external 'some_external_library';

implementation

{$include some_file.inc}
// $I это удобное сокращение от $include, они идентичны
{$I some_other_file.inc}

end.
```

Включаемые файлы обычно имеют расширение `.inc`, и используются для следующих двух целей:

- Включаемый файл может содержать лишь другие директивы компилятора, которые "конфигурируют" исходный код программы. Например, можно создать файл `myconfig.inc` со следующим содержанием:

```
 {$mode objfpc}
 {$H+}
 {$J-}
 {$modeswitch advancedrecords}
 {$ifndef VER3}
   {$error Этот код может быть скомпилирован только в версии FPC не ниже
  3.x.}
```

```
{$endif}
```

Теперь можно включить этот файл `{$I myconfig.inc}` в каждом unit-е исходного кода.

- Второй важный пример использования - разделить unit на несколько файлов, и при этом всё же оставить его одним unit-ом (с учётом всех особенностей Паскаля). Не стоит злоупотреблять таким подходом. В первую очередь стоит думать о том, как разделить программу на несколько unit-ов, а не разделять её на множество include файлов. Однако, в общем, это довольно полезная и удобная техника.

  1. Такой подход позволяет избежать огромного количества unit-ов, и при этом исходные файлы не окажутся слишком длинными. Например, куда удобнее иметь один файл с *"часто используемыми GUI элементами"*, чем создавать *отдельные unit-ы для каждого GUI элемента"*, поскольку второй вариант сделает обычную "uses" неоправданно длинной (поскольку в GUI-приложении будут использоваться множество GUI элементов). Однако, размещение всех этих классов в одном файле `myunit.pas` приведёт к его большому размеру, что в свою очередь затруднит навигацию по файлу. Таким образом, разбить этот файл на несколько include файлов будет разумным.
  2. Такой подход также позволяет легко создавать кросс-платформенный unit с уникальными частями для каждой платформы, что, например, может выглядеть следующим образом:

     ```
     {$ifdef UNIX} {$I my_unix_implementation.inc} {$endif}
     {$ifdef MSWINDOWS} {$I my_windows_implementation.inc} {$endif}
     ```

     Иногда это более удобно, чем писать длинный код со множеством `{$ifdef UNIX}`, `{$ifdef MSWINDOWS}` вперемешку с нормальным кодом (определение переменных, описание процедур). Повышается и читабельность кода. Эту технику можно применять более агрессивно, используя опцию командной строки `-Fi` при вызове FPC, чтобы включить целые папки только для конкретных платформ. Таким образом можно организовать множество версий include файлов типа `{$I my platform_specific_implementation.inc}`. При этом компилятор автоматически найдёт правильную версию.

7.6. Record

Record является своеобразным контейнером для других переменных. Она является очень, очень упрощённым *классом*: в record-ах нет наследования, нет виртуальных методов. Они несколько подобны *structure* в C-подобных языках.

Применяя директиву `{$modeswitch advancedrecords}`, появляется возможность добавить методы и уровни видимости в record и использовать любые особенности языка, доступные классам, которые *не нарушают простую и предсказуемую раскладку памяти для record*.

```
{$mode objfpc}{$H+}{$J-}
{$modeswitch advancedrecords}
type
  TMyRecord = record
  public
    I, Square: Integer;
    procedure WriteLnDescription;
  end;

procedure TMyRecord.WriteLnDescription;
begin
  WriteLn('Квадрат числа ', I, ' равен ', Square);
end;

var
  A: array [0..9] of TMyRecord;
  R: TMyRecord;
  I: Integer;
begin
  for I := 0 to 9 do
  begin
    A[I].I := I;
    A[I].Square := I * I;
  end;

  for R in A do
    R.WriteLnDescription;
end.
```

В современном Паскале, в первую очередь лучше использовать `class`, а не `record`, поскольку классы имеют множество дополнительных полезных возможностей, как конструкторы и наследование. Однако, record-ы всё ещё могут

оказаться полезными если необходима скорость или предсказуемая раскладка памяти:

- Record-ам не требуется constructor или destructor. Они определяются как переменные. Их содержимое не определено в начале (так называемый "мусор в памяти"), кроме случаев автоматически управляемых типов, таких как string, которые всегда инициализируются пустыми. Таким образом следует быть более аккуратным при их использовании, однако преимуществом такого подхода будет увеличение скорости исполнения программы.
- Массивы, состоящие из record-ов однородно располагаются в памяти и таким образом их удобно кешировать.
- Разметка памяти (размер и расстояние между каждым полем) для record-ов чётко определена в некоторых ситуациях: когда запрашивается *C layout* или если используется `packed record`. Это может быть полезным в следующих случаях:
  - для связи между библиотеками, написанными в других языках программирования и предоставляющих API, который основывается на данных типа record,
  - для чтения и записи бинарных файлов,
  - для выполнения "грязных" низкоуровневых оптимизаций (как небезопасное приведение одного типа в другой, на основании их однозначного представления в памяти).
- В record-ах можно также использовать часть `case`, которая работает как *union* в C-подобных языках и позволяет интерпретировать одну и ту же область памяти, как различные типы данных, в зависимости от необходимости. Таким образом можно достичь более высокой эффективности использования памяти в некоторых случаях. В том числе, можно выполнять некоторые "грязные" низкоуровневые оптимизации.

7.7. Устаревшие object

Давным-давно, в Turbo Pascal был введён новый тип синтаксиса с функционалом, похожим на классы, который задавался ключевым словом `object`. По сути, это - нечто среднее между `record` и современным `class`.

- Такие object-ы можно создавать / освобождать, и в процессе этого можно вызвать их constructor / destructor.

- Но их можно просто объявить и использовать как обычную record. Простые типы `record` или `object` не являются указателями (pointer-ами) на что-либо, они, собственно, являются данными. Это делает их весьма удобными в случаях небольших объёмов информации, когда многократное распределение или освобождение памяти не всегда оправдано.
- Устаревшие object-ы имеют наследование и виртуальные методы, впрочем, в несколько отличном виде от современных классов. Следует быть внимательным: если попытаться использовать object, имеющий виртуальные методы, без вызова его constructor-а, могут возникнуть ошибки.

В большинстве случаев крайне не советуется использовать устаревшие objects-ы. Современные *class-ы* имеют гораздо более широкий функционал. А в случае если необходимо повысить скорость выполнения, можно использовать record-ы (включая *advanced records*). Такие подходы обычно лучше, чем устаревшие object-ы.

7.8. Pointer-ы

В Паскале возможно создать *pointer* (указатель) на любой тип данных. Указатель на тип `TMyRecord` определяется с помощью синтаксиса `^TMyRecord`, и чаще всего такие pointer-ы называют `PMyRecord`. В качестве классического примера рассмотрим связанный список целых чисел, организованный с помощью record:

```
type
  PMyRecord = ^TMyRecord;
  TMyRecord = record
    Value: Integer;
    Next: PMyRecord;
  end;
```

Следует обратить внимание, что здесь было использовано рекурсивное определение (тип `PMyRecord` определяется с помощью `TMyRecord`, а `TMyRecord` использует в своём определении `PMyRecord`). Можно определить pointer на тип, *который ещё не был объявлен*, в том случае, если он будет определён в том же самом блоке `type`.

Можно распределять и освобождать память для pointer-ов с помощью методов `New` / `Dispose`, или (более низкоуровневых, но не типобезопасных) методов `GetMem` / `FreeMem`. Для доступа к данным, на которые pointer указывает, после него следует добавить оператор `^` (в виде `PMyRecord^.Value`). Обратная

операция (получить pointer на существующую переменную) выполняется с помощью оператора-префикса `@` (например, `@myRecordVariable`).

Существует также самый общий тип `Pointer`, который не указывает на конкретный тип данных и подобен `void*` в C-подобных языках. Он совершенно не типобезопасен, и его можно привести его тип в любой другой тип pointer-а.

Следует помнить, что *экземпляр класса* также является pointer-ом, хоть для работы с ним и не нужно использовать дополнительных операторов `^` и `@`. Рекурсивные определения также возможны и для классов, и в данном случае всё будет выглядеть ещё проще:

```
type
  TMyClass = class
    Value: Integer;
    Next: TMyClass;
  end;
```

7.9. Перегрузка операторов

В Паскале существует возможность "перегрузить" (overload) значение многих операторов языка, например определить сложение и умножение пользовательских типов данных. Рассмотрим следующий пример:

```
{$mode objfpc}{$H+}{$J-}
uses StrUtils;

operator* (const S: string; const A: Integer): string;
begin
  Result := DupeString(S, A);
end;

begin
  WriteLn('повтор' * 10);
end.
```

Можно также перегружать операторы над классами. Учитывая то, что обычно в такой функции-операторе создаётся новый экземпляр класса, вызывающий код должен позаботиться об надлежащем освобождении памяти.

```
{$mode objfpc}{$H+}{$J-}
```

```
uses
  SysUtils;

type
  TMyClass = class
    MyInt: Integer;
  end;

operator* (const C1, C2: TMyClass): TMyClass;
begin
  Result := TMyClass.Create;
  Result.MyInt := C1.MyInt * C2.MyInt;
end;

var
  C1, C2: TMyClass;
begin
  C1 := TMyClass.Create;
  try
    C1.MyInt := 12;
    C2 := C1 * C1;
    try
      WriteLn('12 * 12 = ', C2.MyInt);
    finally
      FreeAndNil(C2);
    end;
  finally
    FreeAndNil(C1);
  end;
end.
```

Можно перегружать и операции над record-ами. Обычно это проще, чем в случае классов, поскольку нет необходимости выполнять операции по распределению или освобождению памяти.

```
{$mode objfpc}{$H+}{$J-}
uses
  SysUtils;

type
  TMyRecord = record
    MyInt: Integer;
  end;
```

```
operator* (const C1, C2: TMyRecord): TMyRecord;
begin
  Result.MyInt := C1.MyInt * C2.MyInt;
end;

var
  R1, R2: TMyRecord;
begin
  R1.MyInt := 12;
  R2 := R1 * R1;
  WriteLn('12 * 12 = ', R2.MyInt);
end.
```

При работе с record-ами лучше использовать режим `{$modeswitch advancedrecords}` и перегружать операторы в виде `class operator` внутри объявления record-а. Такой подход позволяет использовать generic классы которые зависят от существования какого-либо оператора (например `TFPGList`, который зависит от доступности оператора равенства). В противоположном случае "глобальное" определение оператора (не внутри объявления данной record) не будет найдено (поскольку оно не доступно коду, который используется в `TFPGList`), и не удастся specialize список в виде `specialize TFPGList<TMyRecord>`.

```
{$mode objfpc}{$H+}{$J-}
{$modeswitch advancedrecords}
uses
  SysUtils, FGL;

type
  TMyRecord = record
    MyInt: Integer;
    class operator+ (const C1, C2: TMyRecord): TMyRecord;
    class operator= (const C1, C2: TMyRecord): boolean;
  end;

class operator TMyRecord.+ (const C1, C2: TMyRecord): TMyRecord;
begin
  Result.MyInt := C1.MyInt + C2.MyInt;
end;

class operator TMyRecord.= (const C1, C2: TMyRecord): boolean;
begin
  Result := C1.MyInt = C2.MyInt;
```

```
end;

type
  TMyRecordList = specialize TFPGList<TMyRecord>;

var
  R, ListItem: TMyRecord;
  L: TMyRecordList;
begin
  L := TMyRecordList.Create;
  try
    R.MyInt := 1;   L.Add(R);
    R.MyInt := 10;  L.Add(R);
    R.MyInt := 100; L.Add(R);

    R.MyInt := 0;
    for ListItem in L do
      R := ListItem + R;

    WriteLn('1 + 10 + 100 = ', R.MyInt);
  finally
    FreeAndNil(L);
  end;
end.
```

8. Дополнительные возможности классов

8.1. Различие private и strict private

Спецификатор видимости `private` означает что поле или метод не доступны извне данного класса. Однако из данного правила есть одно исключение: любой код *в данном unit-е* может преодолеть это ограничение и иметь полный доступ к private полям и методам. Программист на C++ сказал бы, что в Паскале *все классы в одном unit-е являются "друзьями"*. Часто это весьма удобно.

Однако, при создании unit-ов большого размера со множеством не очень близко интегрированных классов, более безопасным является применение видимости `strict private`. Как можно легко догадаться, это означает что поле или метод не доступны извне данного класса. Точка. Никаких исключений.

Таким же образом различаются спецификаторы видимости `protected` (видимый наследникам данного класса или "друзьям" в том же unit-е) и `strict`

`protected` (видимы только и исключительно наследникам данного класса. Точка).

8.2. Class method

Существуют методы, которые можно вызвать из класса вообще (`TMyClass`), не обязательно из его конкретного экземпляра.

```
type
  TEnemy = class
    procedure Kill;
    class procedure KillAll;
  end;

var
  E: TEnemy;
begin
  E := TEnemy.Create;
  try
    E.Kill;
  finally
    FreeAndNil(E);
  end;
  TEnemy.KillAll;
end;
```

Обратите внимание, что они также могут быть виртуальными — иногда это весьма удобно, особенно, если применяется совместно с понятием Section 8.4, “Ссылки на класс”.

Следует отметить, что constructor всегда работает, как class method когда вызывается "обычным образом" (`MyInstance := TMyClass.Create(…);`). Впрочем, можно вызвать конструктор в данном конкретном экземпляре класса как метод — и он тогда и сработает как обычный метод. Таким образом можно сделать удобную "цепочку" constructor-ов, когда один constructor (например, перегруженный для принятия дополнительных параметров) выполняет определённый код, а затем вызывает другой constructor (например, беспараметрический).

8.3. Дополнительные возможности объявления классов и локальные классы

Внутри определения класса можно создавать константы (`const`) или типы (`type`). Таким образом можно даже создать класс внутри класса. Спецификаторы видимости будут работать как обычно, и такие локальные классы могут быть private (не видимые "внешнему миру"), что иногда удобно.

Следует обратить внимание, чтобы определить поле после константы или типа будет необходимо использовать блок `var`:

```
type
  TMyClass = class
  private
    type
      TInternalClass = class
        Velocity: Single;
        procedure DoSomething;
      end;
    var
      FInternalClass: TInternalClass;
  public
    const
      DefaultVelocity = 100.0;
    constructor Create;
    destructor Destroy; override;
  end;

constructor TMyClass.Create;
begin
  inherited;
  FInternalClass := TInternalClass.Create;
  FInternalClass.Velocity := DefaultVelocity;
  FInternalClass.DoSomething;
end;

destructor TMyClass.Destroy;
begin
  FreeAndNil(FInternalClass);
  inherited;
end;

{ Обратите внимание на префикс "TMyClass.TInternalClass." }
```

```
procedure TMyClass.TInternalClass.DoSomething;
begin
end;
```

8.4. Ссылки на класс

Ссылки на класс позволяют выбрать класс в процессе исполнения программы, например, для вызова class method-а или constructor-а не определив заранее, какой именно класс будет использоваться. Такой тип объявляется следующим образом: `class of TMyClass`.

```
type
  TMyClass = class(TComponent)
  end;

  TMyClass1 = class(TMyClass)
  end;

  TMyClass2 = class(TMyClass)
  end;

  TMyClassRef = class of TMyClass;

var
  C: TMyClass;
  ClassRef: TMyClassRef;
begin
  // Можно сделать так:

  C := TMyClass.Create(nil); FreeAndNil(C);
  C := TMyClass1.Create(nil); FreeAndNil(C);
  C := TMyClass2.Create(nil); FreeAndNil(C);

  // А с помощью ссылки на класс можно сделать следующим образом:

  ClassRef := TMyClass;
  C := ClassRef.Create(nil); FreeAndNil(C);

  ClassRef := TMyClass1;
  C := ClassRef.Create(nil); FreeAndNil(C);

  ClassRef := TMyClass2;
  C := ClassRef.Create(nil); FreeAndNil(C);
```

```
end;
```

Ссылки на класс можно комбинировать с виртуальными class method-ами. Работает это подобно классам с виртуальными методами — конкретный метод, который необходимо выполнить определяется уже в процессе выполнения программы.

```
type
  TMyClass = class(TComponent)
    class procedure DoSomething; virtual; abstract;
  end;

  TMyClass1 = class(TMyClass)
    class procedure DoSomething; override;
  end;

  TMyClass2 = class(TMyClass)
    class procedure DoSomething; override;
  end;

  TMyClassRef = class of TMyClass;

var
  C: TMyClass;
  ClassRef: TMyClassRef;
begin
  ClassRef := TMyClass1;
  ClassRef.DoSomething;

  ClassRef := TMyClass2;
  ClassRef.DoSomething;

  { А следующая строка приведёт к ошибке выполнения,
    поскольку DoSomething является abstract в TMyClass. }
  ClassRef := TMyClass;
  ClassRef.DoSomething;
end;
```

Если есть экземпляр класса и необходимо создать ссылку на этот класс (не на какой-либо объявленный класс, а на конкретного наследника, который был использован при создании данного экземпляра класса), можно использовать свойство `ClassType`. Вообще говоря, объявленный тип `ClassType` является `TClass`, который в свою очередь является `class of TObject`. Его тип можно

привести во что либо более конкретное, когда есть информация, чем именно является данный экземпляр.

`ClassType` также можно использовать для вызова виртуальных методов, включая виртуальные constructor-ы. Такой подход позволяет создать такие методы, как `Clone` которые создают экземпляр *как точную копию данного класса в конкретный момент исполнения программы*. Можно совместить такой подход с Assign (см. Section 6.3, “Клонирование классов: TPersistent.Assign”) для того, чтобы метод возвращал новую, готовую к работе копию текущего экземпляра.

Следует обратить внимание, что подобный подход сработает только если constructor данного класса виртуальный. Например, его можно использовать с наследниками стандартного класса `TComponent`, поскольку все они выполняют override виртуального constructor-а `TComponent.Create(AOwner: TComponent)`.

```
type
  TMyClass = class(TComponent)
    procedure Assign(Source: TPersistent); override;
    function Clone(AOwner: TComponent): TMyClass;
  end;

  TMyClassRef = class of TMyClass;

function TMyClass.Clone(AOwner: TComponent): TMyClass;
begin
  // Таким образом будет создан объект конкретного типа TMyClass:
  //Result := TMyClass.Create(AOwner);
  // А такой подход может создать объект как типа TMyClass, так и его
 наследников:
  Result := TMyClassRef(Self.ClassType).Create(AOwner);
  Result.Assign(Self);
end;
```

8.5. Class helper

Метод является лишь процедурой внутри конкретного класса. Извне класса он вызывается с помощью специального синтаксиса `MyInstance.MyMethod(…)`. И через некоторое время приходит привычка, что *если нужно произвести действие над классом X, следует писать* `X.Action(…)`.

Однако, иногда возникает необходимость выполнить что-либо, что концептуально является *действием на класс TMyClass*, однако при этом не изменяя исходный код *TMyClass*. Причин тому может быть несколько. Например, это может быть исходный код, написанный другим программистом, который не следует или невозможно изменять. Также иногда причиной тому могут быть зависимости — добавление метода `Render` к классу `TMy3DObject` кажется вполне логичным, однако, возможно, имплементация класса `TMy3DObject` должна быть независимой от кода рендера? В таких случаях удобнее "расширить" существующий класс, добавив к нему функционал, при этом не изменяя его исходный код.

Наиболее простой путь сделать это - создать глобальную процедуру, которая будет принимать ссылку на `TMy3DObject` как параметр.

```
procedure Render(const Obj1: TMy3DObject; const Color: TColor);
var
  I: Integer;
begin
  for I := 0 to Obj1.ShapesCount - 1 do
    RenderMesh(Obj1.Shape[I].Mesh, Color);
end;
```

И это действительно сработает. Однако, недостаток такого подхода - он выглядит не очень красиво. Ведь обычно мы вызываем действия над объектом с помощью `X.Action(…)`, а тут нам приходится использовать иной синтаксис: `Render(X, …)`. Было бы куда удобнее записать `X.Render(…)`, даже для случаев, когда `Render` не описан в unit-е, в котором находится `TMy3DObject`.

Для этого и существуют class helper-ы, дающие возможность создать процедуры/функции, которые оперируют данным классом и вызываются таким же образом, как и остальные методы класса. Однако они не являются "обычными" методами — они "добавляются" извне определения класса `TMy3DObject`.

```
type
  TMy3DObjectHelper = class helper for TMy3DObject
    procedure Render(const Color: TColor);
  end;

procedure TMy3DObjectHelper.Render(const Color: TColor);
var
  I: Integer;
```

```
begin
  { Обратите внимание, мы получаем доступ к ShapesCount, Shape без
 дополнительных указаний типа TMy3DObject.ShapesCount }
  for I := 0 to ShapesCount - 1 do
    RenderMesh(Shape[I].Mesh, Color);
end;
```

ПРИМЕЧАНИЕ: Более общая концепция является *"type helper"*, используя которую становится возможным добавлять методы даже к самым примитивным типам, например integer. Можно также создать *"record helper"* чтобы… ну, Вы поняли. Детальнее см. здесь: http://lists.freepascal.org/fpc-announce/2013-February/000587.html .

8.6. Виртуальные constructor-ы, destructor-ы

Имя destructor-а всегда должно быть `Destroy`, и он всегда является virtual (поскольку он вызывается без указания конкретного класса при компиляции) и беспараметрический.

В качестве имени constructor-а принято использовать `Create`. Можно использовать и другое имя, однако здесь следует быть аккуратным — например, создав `CreateMy`, всегда создайте и `Create`, иначе constructor `Create` будет всё равно доступен из родительского класса и может быть вызван в обход конкретного `CreateMy` конструктора.

В базовом классе `TObject` constructor не является виртуальным, и при создании наследников его можно изменить. Новый constructor просто спрячет constructor родительского класса (примечание: не используйте `overload`, в данном случае это не сработает).

В наследниках же класса `TComponent`, следует выполнять `constructor Create(AOwner: TComponent); override;`. Для решения задач потоковой передачи данных, а также для создания класса без указания конкретного типа при написании программы, виртуальные constructor-ы являются очень удобными (см. раздел Section 8.4, “Ссылки на класс” выше).

8.7. Ошибки при исполнении constructor-а

Что произойдёт если в процессе выполнения constructor-а возникнет ошибка? Строка

```
X := TMyClass.Create;
```

не будет выполнена до конца, и `X` не может быть присвоено какое-либо значение. Кто будет выполнять очистку после частично сконструированного класса?

В объектном Паскале существует следующее решение. Если возникла ошибка при исполнении constructor-а, то сразу вызывается destructor. Именно по этой причине *этот destructor должен быть "дубовым"*, т.е. сработать в любом случае, даже на частично сконструированном классе. Обычно это не сложно, если освобождать всё безопасным образом, например, с помощью `FreeAndNil`.

Также можно полагаться на факт, что перед вызовом constructor-а *вся память гарантированно обнуляется*. Таким образом, при создании все ссылки внутри класса являются `nil`, а числа равны 0 и т.п.

Т.е. следующий код сработает без утечек памяти:

```
{$mode objfpc}{$H+}{$J-}
uses
  SysUtils;

type
  TGun = class
  end;

  TPlayer = class
    Gun1, Gun2: TGun;
    constructor Create;
    destructor Destroy; override;
  end;

constructor TPlayer.Create;
begin
  inherited;
  Gun1 := TGun.Create;
  raise Exception.Create('Вызваем exception из constructor-а!');
  Gun2 := TGun.Create;
end;

destructor TPlayer.Destroy;
begin
  { В данном случае в результате ошибки в constructor-е, у нас
    может оказаться Gun1 <> nil и Gun2 = nil. Смиритесь.
```

```
    В таком случае, FreeAndNil справится с задачей без каких-либо
    дополнительных действий с нашей стороны, поскольку FreeAndNil
 проверяет
    является ли экземпляр класса nil перед вызовом соответствующего
 destructor-a. }
  FreeAndNil(Gun1);
  FreeAndNil(Gun2);
  inherited;
end;

begin
  try
    TPlayer.Create;
  except
    on E: Exception do
      WriteLn('Ошибка ' + E.ClassName + ': ' + E.Message);
  end;
end.
```

9. Интерфейсы

9.1. Хорошие (CORBA) интерфейсы

Интерфейс, так же как и класс, объявляет API, но не определяет его конкретную реализацию. Класс может иметь только один родительский класс, однако может реализовать множество интерфейсов.

Можно выполнить приведение типа класса к любому интерфейсу, которому он соответствует, и потом *вызывать его методы через этот интерфейс*. Это позволяет унифицированным образом обрабатывать множество классов, которые не наследуют друг друга, однако имеют подобный функционал. Такой подход полезен в случае, если недостаточно простого механизма наследования.

Интерфейсы CORBA в Object Pascal работают в значительной степени так, как и интерфейсы в Java (https://docs.oracle.com/javase/tutorial/java/concepts/interface.html) или в C# (https://msdn.microsoft.com/en-us/library/ms173156.aspx).

```
{$mode objfpc}{$H+}{$J-}
{$interfaces corba}

uses
  SysUtils, Classes;
```

```
type
  IMyInterface = interface
  ['{79352612-668B-4E8C-910A-26975E103CAC}']
    procedure Shoot;
  end;

  TMyClass1 = class(IMyInterface)
    procedure Shoot;
  end;

  TMyClass2 = class(IMyInterface)
    procedure Shoot;
  end;

  TMyClass3 = class
    procedure Shoot;
  end;

procedure TMyClass1.Shoot;
begin
  WriteLn('TMyClass1.Shoot');
end;

procedure TMyClass2.Shoot;
begin
  WriteLn('TMyClass2.Shoot');
end;

procedure TMyClass3.Shoot;
begin
  WriteLn('TMyClass3.Shoot');
end;

procedure UseThroughInterface(I: IMyInterface);
begin
  Write('Стреляем... ');
  I.Shoot;
end;

var
  C1: TMyClass1;
  C2: TMyClass2;
  C3: TMyClass3;
begin
  C1 := TMyClass1.Create;
```

```
  C2 := TMyClass2.Create;
  C3 := TMyClass3.Create;
  try
    if C1 is IMyInterface then
      UseThroughInterface(C1 as IMyInterface);
    if C2 is IMyInterface then
      UseThroughInterface(C2 as IMyInterface);
    if C3 is IMyInterface then
      UseThroughInterface(C3 as IMyInterface);
  finally
    FreeAndNil(C1);
    FreeAndNil(C2);
    FreeAndNil(C3);
  end;
end.
```

9.2. CORBA и COM интерфейсы

Почему интерфейсы названы "CORBA"?

Название **CORBA** крайне неудачное. Куда лучший термин был бы **обычный интерфейс**. Такие интерфейсы являются *"чистой особенностью языка"*. Их можно использовать, если возникает необходимость приведения типов различных классов в виде одинакового интерфейса, поскольку у них одинаковая API.

Не смотря на то, что эти интерфейсы могут использоваться совместно с *технологией CORBA (Common Object Request Broker Architecture)* (см. https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture), они *не* имеют к ней никакого отношения.

Требуется ли задавать `{$interfaces corba}` ?

Да, поскольку по умолчанию будут созданы *COM интерфейсы*. Последнее можно указать явно с помощью директивы `{$interfaces com}`, но не обязательно, так как это случай по умолчанию.

Лично я не советую использовать *COM интерфейсы*. Особенно в случае, если Вам знакомы интерфейсы из других языков программирования. *Интерфейсы CORBA* в Паскале — это именно то, что ожидается от интерфейсов, они идентичны интерфейсам в C# и Java. При этом *COM интерфейсы* привносят дополнительные "особенности", которые, скорее всего, не понадобятся.

Обратите внимание, что `{$interfaces xxx}` определяет только интерфейсы, родительский интерфейс которых явно не указан (лишь используется ключевое слово `interface`, а не `interface(ISomeAncestor)`). Если интерфейс имеет родительский интерфейс, он имеет такой же тип, как и родительский интерфейс и не зависит от директивы `{$interfaces xxx}`.

Что такое COM интерфейсы?

Понятие *COM интерфейс* является синонимом понятия *интерфейс, наследующий некоторый интерфейс* `IUnknown`. Такое наследование от `IUnknown`:

- Требует, чтобы классы определяли методы `_AddRef` и `_ReleaseRef`. Правильная имплементация этих методов позволит управлять объектом с помощью подсчёта ссылок (reference-counting).
- Добавляет метод `QueryInterface`.
- Позволяет взаимодействовать с технологией *COM (Компонентная модель объектов)*.

Почему я не советую использовать COM интерфейсы?

Дополнительные возможности, привнесённые COM интерфейсами с моей точки зрения проблематичны. Не поймите превратно — идея reference-counting очень хороша. Однако переплетение такого функционала с интерфейсами (вместо того, чтобы делать их "поперечными" свойствами), на мой взгляд, является очень грязным. И явно не соответствует задачам, которые я встречал в своей практике.

- Иногда возникает необходимость передать обычные классы (не имеющими ничего общего) через обычный интерфейс.
- Иногда возникает желание управлять памятью с помощью технологии reference counting.
- *Возможно* когда-нибудь лично мне тоже пригодится *COM технология*.

Но всё это - ничем не связанные задачи. Переплетать их в единой особенности языка, на мой личный взгляд, нехорошо. И это не просто вопрос эстетики, такой подход причиняет реальные проблемы: механизм reference-counting интерфейса COM, даже если отключён с помощью специальных имплементаций `_AddRef` и `_ReleaseRef`, всё равно может привести к ошибкам. Придётся быть внимательным и следить, чтобы нигде не осталась временная ссылка на интерфейс после освобождения экземпляра класса. Чуть подробнее речь об этом пойдёт дальше.

Именно по этому, мой совет: всегда использовать интерфейс в стиле *CORBA* и соответствующую `{$interfaces corba}` директиву в современном коде, который работает с интерфейсами. На мой взгляд, *COM* интерфейсы это некоторое *"недоразумение" языка*.

Однако, чтобы быть честными, чуть дальше будет детально идти речь и о *COM интерфейсах*.

Можно ли использовать reference-counting совместно с интерфейсом CORBA?

Естественно. Необходимо лишь добавить методы `_AddRef` / `_ReleaseRef`. Нет необходимости наследовать интерфейс `IUnknown`. Впрочем, в большинстве случаев, если возникает необходимость использовать reference-counting в интерфейсах, можно просто использовать COM интерфейсы.

9.3. GUID-ы интерфейсов

GUID-ы это на первый взгляд случайная последовательность букв и цифр `['{ABCD1234-…}']` которую можно увидеть при каждом объявлении интерфейса. В действительности, они и являются случайными. Но, к сожалению, в них есть необходимость. Никакого особого смысла в них не вкладывается (если не планируется интеграция с технологиями *COM* или *CORBA*). Однако, для правильного исполнения программы они обязательны. И пусть не сбивает с толку компилятор, который, увы, позволяет создавать интерфейсы без GUID-ов.

Без присваивания (уникального) GUID-а, все интерфейсы будут идентичными для оператора `is`. Таким образом, он всегда будет возвращать `true` если данный класс поддерживает *любой* из используемых интерфейсов. "Волшебная" функция `Supports(ObjectInstance, IMyInterface)` работает несколько лучше в данном случае, поскольку выдаст ошибку компиляции для интерфейсов без GUID. Это касается и CORBA, и COM интерфейсов, для версии FPC 3.0.0.

Таким образом, необходимо обязательно объявлять GUID для каждого интерфейса. Можно использовать встроенный генератор GUID-ов *Lazarus GUID generator* (горячая клавиша `Ctrl + Shift + G` в режиме редактирования). Либо воспользоваться он-лайн сервисом, например https://www.guidgenerator.com/ .

А можно вообще написать свой инструмент, использующий `CreateGUID` и `GUIDToString` функции из RTL. Например, следующим образом:

```
{$mode objfpc}{$H+}{$J-}
```

```
uses
  SysUtils;
var
  MyGuid: TGUID;
begin
  Randomize;
  CreateGUID(MyGuid);
  WriteLn('[''' + GUIDToString(MyGuid) + ''']');
end.
```

9.4. Некрасивые (COM) интерфейсы

Использование *COM интерфейсов* даёт две дополнительных возможности:

1. Интеграция с технологией COM (технология, которая используется в Windows, также доступна в Unix-системах через *XPCOM*, который применяется Mozilla),
2. Подсчёт ссылок - reference counting (что позволяет автоматически освобождать экземпляр класса, когда все ссылки на его интерфейс уходят из поля переменных).

На мой взгляд, неправильно переплетать *интерфейсы* с такими возможностями. Это усложняет использование интерфейсов для того, для чего они предназначены: когда у многих классов один и тот же API, однако они не могут происходить от одного родительского класса. Используя же *COM интерфейс*, следует помнить о механизме *автоматического уничтожения* и его отношении к технологии COM.

На практике это означает следующее:

- В классе необходимо создать "волшебные" методы `_AddRef`, `_Release` и `QueryInterface`. Или наследовать класс, который уже имеет их. Конкретная имплеметация этих методов позволяет включить или отключить такую возможность COM интерфейсов, как *reference-counting*. Впрочем, отключать её достаточно опасно — см. следующий подраздел.
  - Стандартный класс `TInterfacedObject` имеет эти методы выполненными в таком виде, чтобы *включить* reference-counting.
  - Стандартный класс `TComponent` имеет эти методы выполненными в таком виде, чтобы *выключить* reference-counting. В **Castle Game Engine** предоставлены дополнительные классы, от которых можно выполнять наследование: `TNonRefCountedInterfacedObject`

и `TNonRefCountedInterfacedPersistent` для этих целей, см. детальнее: https://github.com/castle-engine/castle-engine/blob/master/src/base/castleinterfaces.pas .

- Необходимо быть аккуратным при освобождении класса, когда на него могут ссылаться некоторые переменные интерфейса. Поскольку интерфейсы освобождаются с помощью виртуального метода (потому что он *может использовать reference-counting, даже если* `_AddRef` *написан таким образом, чтобы отключить эту возможность*), то нельзя освобождать низлежащий экземпляр класса из-за того, что какая-либо переменная интерфейса может на него указывать. См. раздел *"7.7 Reference counting"* в руководстве FPC (http://freepascal.org/docs-html/ref/refse47.html).

Чтобы безопасно использовать *COM интерфейсы* необходимо

- осознавать факт, что в них используется reference-counting,
- наследовать соответствующие классы от `TInterfacedObject`
- и избегать прямого обращения к экземпляру класса, вместо чего всегда использовать экземпляр через интерфейс, оставляя алгоритму reference-counting управление освобождением памяти.

Ниже представлен пример использования такого интерфейса:

```
{$mode objfpc}{$H+}{$J-}
{$interfaces com}

uses
  SysUtils, Classes;

type
  IMyInterface = interface
  ['{3075FFCD-8EFB-4E98-B157-261448B8D92E}']
    procedure Shoot;
  end;

  TMyClass1 = class(TInterfacedObject, IMyInterface)
    procedure Shoot;
  end;

  TMyClass2 = class(TInterfacedObject, IMyInterface)
    procedure Shoot;
  end;
```

```
  TMyClass3 = class(TInterfacedObject)
    procedure Shoot;
  end;

procedure TMyClass1.Shoot;
begin
  WriteLn('TMyClass1.Shoot');
end;

procedure TMyClass2.Shoot;
begin
  WriteLn('TMyClass2.Shoot');
end;

procedure TMyClass3.Shoot;
begin
  WriteLn('TMyClass3.Shoot');
end;

procedure UseThroughInterface(I: IMyInterface);
begin
  Write('Стреляем... ');
  I.Shoot;
end;

var
  C1: IMyInterface;  // COM управляет освобождением памяти
  C2: IMyInterface;  // COM управляет освобождением памяти
  C3: TMyClass3;     // Здесь управлять освобождением памяти придётся ВАМ
begin
  C1 := TMyClass1.Create as IMyInterface;
  C2 := TMyClass2.Create as IMyInterface;
  C3 := TMyClass3.Create;
  try
    UseThroughInterface(C1); // Нет необходимости в операторе "as"
    UseThroughInterface(C2);
    if C3 is IMyInterface then
      UseThroughInterface(C3 as IMyInterface); // Так не сработает
  finally
    { Переменные C1 и C2 выходят из поля зрения
      и будут автоматически уничтожены сейчас.

      а переменная C3 является экземпляром класса
      и не управляется интерфейсом,
```

```
      и по этому её необходимо совободить вручную. }
    FreeAndNil(C3);
  end;
end.
```

9.5. Использование COM интерфейсов с отключённым reference-counting

Как уже было упомянуто в прошлом разделе, в классах, наследующих `TComponent` (либо подобные классы, такие как `TNonRefCountedInterfacedObject` и `TNonRefCountedInterfacedPersistent`) отключено reference-counting для COM интерфейсов. Это позволяет использовать COM интерфейсы и при этом даёт возможность освобождать классы вручную.

Всё же, необходимо быть аккуратным в этом случае, чтобы не освободить класс, когда какая-либо переменная интерфейса ссылается на него и не забывать, что каждая операция приведения типа в виде `Cx as IMyInterface` также создаёт временную переменную интерфейса, которая может существовать вплоть до конца текущей процедуры. Этот пример иллюстрирует процедуру `UseInterfaces`, которая освобождает классы *вне* этой процедуры (когда мы можем быть уверены, что временная переменная интерфейса вышла из текущего поля переменных).

Чтобы избежать этих неудобств, лучше использовать интерфейсы CORBA, если в данной программе нет необходимости параллельного использования reference-counting и интерфейсов.

```
{$mode objfpc}{$H+}{$J-}
{$interfaces com}

uses
  SysUtils, Classes;

type
  IMyInterface = interface
  ['{3075FFCD-8EFB-4E98-B157-261448B8D92E}']
    procedure Shoot;
  end;

  TMyClass1 = class(TComponent, IMyInterface)
    procedure Shoot;
```

```
  end;

  TMyClass2 = class(TComponent, IMyInterface)
    procedure Shoot;
  end;

  TMyClass3 = class(TComponent)
    procedure Shoot;
  end;

procedure TMyClass1.Shoot;
begin
  WriteLn('TMyClass1.Shoot');
end;

procedure TMyClass2.Shoot;
begin
  WriteLn('TMyClass2.Shoot');
end;

procedure TMyClass3.Shoot;
begin
  WriteLn('TMyClass3.Shoot');
end;

procedure UseThroughInterface(I: IMyInterface);
begin
  Write('Стреляем... ');
  I.Shoot;
end;

var
  C1: TMyClass1;
  C2: TMyClass2;
  C3: TMyClass3;

procedure UseInterfaces;
begin
  if C1 is IMyInterface then
  //if Supports(C1, IMyInterface) then // эта строчка идентична проверке
 "is" выше
    UseThroughInterface(C1 as IMyInterface);
  if C2 is IMyInterface then
    UseThroughInterface(C2 as IMyInterface);
  if C3 is IMyInterface then
```

```
    UseThroughInterface(C3 as IMyInterface);
end;

begin
  C1 := TMyClass1.Create(nil);
  C2 := TMyClass2.Create(nil);
  C3 := TMyClass3.Create(nil);
  try
    UseInterfaces;
  finally
    FreeAndNil(C1);
    FreeAndNil(C2);
    FreeAndNil(C3);
  end;
end.
```

9.6. Приведение типов интерфейсов без оператора "as"

Этот раздел касается как *CORBA*, так и *COM* интерфейсов.

Приведение типа интерфейса через оператор `as` выполняет проверку в режиме исполнения. Рассмотрим пример:

```
UseThroughInterface(Cx as IMyInterface);
```

Этот вариант скомпилируется для всех экземпляров `C1`, `C2`, `C3` из примера в предыдущем подразделе. При выполнении программы возникнет ошибка для `C3`, у которого не описан `IMyInterface` (впрочем, мы можем избежать такой ошибки, проверив `Cx is IMyInterface` перед выполнением приведения типа), как это сделано в примере.

Однако, возможно привести тип данного экземпляра самым непосредственным образом:

```
UseThroughInterface(Cx);
```

В таком случае проверка выполняется в момент компиляции. Такой код скомпилируется для `C1` и `C2` (которые определены, как классы, использующие `IMyInterface`), но не скомпилируется для `C3`.

По большому счёту, такое приведение типа работает как для обычных классов. В случае, если требуется экземпляр класса `TMyClass`, всегда можно

использовать переменную, объявленную как `TMyClass`, **которой подойдёт любой наследник** `TMyClass` Точно такая же ситуация и для интерфейсов. Не требуется выполнять явные приведения типов в такой ситуации.

Эквивалентно этому:

```
UseThroughInterface(IMyInterface(Cx));
```

Это тоже приведение типа, которое должно быть верным при компиляции. Обратите внимание, что данный синтаксис работает иначе, чем приведение типов обычных классов. Если написать что-либо вроде `TMyClass(C)` для классов, то в результате получится *небезопасное, непроверяемое приведение типов*. В случае же интерфейсов, писать `IMyInterface(C)` безопасно и быстро (проверяется лишь при компиляции).

10. Про этот документ

Исходные файлы этого документа в формате AsciiDoc можно скачать по адресу: https://github.com/michaliskambi/modern-pascal-introduction . Автор будет рад любым пожеланиям, исправлениям, расширениям материала, доработкам и pull request-ам :). С автором можно связаться через профиль GitHub либо по e-mail: michalis.kambi@gmail.com[1] . Домашняя страничка автора: https://michalis.ii.uni.wroc.pl/~michalis/ .

Благодарю за прочтение!

Перевод на русский выполнен: Александр Скворцов и Евгений Лоза 2016-2017

[1] mailto:michalis.kambi@gmail.com