Модерен обектен Паскал - Въведение за програмисти

Michalis Kamburelis

Съдържание

1. Защо . . . 1
2. Основи . . . 1
   2.1. Програма "Hello world" . . . 1
   2.2. Функции, процедури, примитивни типове . . . 2
   2.3. Проверки (if) . . . 4
   2.4. Логически, релационни и побитови оператори . . . 5
   2.5. Тестване на единичен израз за множество стойности (case) . . . 6
   2.6. Изброени и бройни типове, множества и масиви с постоянна дължина . . . 7
   2.7. Цикли (for, while, repeat, for .. in) . . . 8
   2.8. Изпечатване на информация, логове . . . 11
   2.9. Преобразуване в низ . . . 12
3. Модули (Unit-и) . . . 13
   3.1. Unit-и, които се използват взаимно . . . 14
   3.2. Квалифициране на идентификаторите с името на unit-а . . . 16
   3.3. Представяне на идентификаторите от един unit чрез друг . . . 18
4. Класове . . . 19
   4.1. Основи . . . 19
   4.2. Наследяване, проверка (is), конверсия на типа (as) . . . 20
   4.3. Свойства . . . 22
      4.3.1. Сериализация на свойства . . . 24
   4.4. Изключения - Кратък пример . . . 25
   4.5. Нива на видимост . . . 26
   4.6. Предшественик по подразбиране . . . 26
   4.7. Self . . . 27
   4.8. Извикване на наследен метод . . . 27
   4.9. Виртуални методи, подмяна и скриване . . . 30
5. Освобождаване на паметта за класове . . . 33
   5.1. Помнете да освобождавате паметта заета от инстанциите . . . 33
   5.2. Как да освободим паметта . . . 33
   5.3. Ръчно и автоматично освобождаване . . . 34
   5.4. Виртуалният деструктор Destroy . . . 37
   5.5. Известие при освобождаване . . . 38
   5.6. Наблюдател за известие при освобождаване (Castle Game Engine) . . . 40
6. Изключения . . . 41
   6.1. Преглед . . . 41
   6.2. Предизвикване . . . 42
   6.3. Прихващане . . . 43
   6.4. Finally (изпълнение на код независимо дали има изключение) . . . 45

6.5. Как изключенията се показват от различните библиотеки . . . 48
7. Run-time библиотека . . . 48
7.1. Вход/изход с помощта на потоци . . . 48
7.2. Контейнери (списъци, речници), използващи генерици . . . 50
7.3. Клониране: TPersistent.Assign . . . 55
8. Разни възможности на езика . . . 59
8.1. Локални (вложени) подпрограми . . . 59
8.2. Callbacks (познати като *Събития*, също като *Указатели към функции*, също като *Процедурни променливи*) . . . 60
8.3. Генерици . . . 63
8.4. Overloading . . . 65
8.5. Препроцесор . . . 65
8.6. Записи . . . 67
8.7. Обекти, стар стил . . . 69
8.8. Указатели . . . 69
8.9. Замяна на оператори . . . 70
9. Допълнителни възможности на класовете . . . 74
9.1. Частни и лични полета . . . 74
9.2. Допълнителни декларации и вложени класове . . . 74
9.3. Методи на класа . . . 75
9.4. Препратки към клас . . . 76
9.5. Статични методи на клас . . . 78
9.6. Полета и свойства на клас . . . 79
9.7. Помощници за клас . . . 80
9.8. Виртуални конструктори, деструктори . . . 81
9.9. Изключение в конструктор . . . 82
10. Интерфейси . . . 83
10.1. Голи (CORBA) интерфейси . . . 83
10.2. Интерфейси CORBA и COM . . . 85
10.3. Интерфейсни GUIDs . . . 87
10.4. Интерфейси с броене на препратки (COM) . . . 88
10.5. Използване на COM интерфейси с изключено броене . . . 90
10.6. Преобразуване на интерфейси . . . 92
11. Относно този документ . . . 95

1. Защо

Има много книги и ресурси за Паскал, но в повечето от тях се говори само за стария Паскал, който е без класове, модули[1] или генерици[2].

Затова написах кратко въведение в това, което аз наричам **модерен Обектен Паскал**. Повечето от програмистите, които го използват, всъщност не го наричат така. Просто го наричаме *"нашия Паскал"*. Но чувствам, че когато представям езика е важно да подчертая, че е вече модерен, обектно-ориентиран език. Той се е развил **значително** от времето на стария (Turbo) Паскал, който много хора са учили преди време в училище. Функционално е доста подобен на C++, Java или C#.

- Той има всички съвременни функции, които можете да очаквате - класове, модули, интерфейси[3], генерици ...
- Той се компилира директно до бърз машинен код,
- Той е типово обезопасен,
- Той е език от високо ниво, но също така може да е и от ниско ако желаете.

Освен това има отличен преносим компилатор с отворен код, наречен *Free Pascal Compiler*, http://freepascal.org/ . Има и придружаващо IDE (редактор, Debugger, библиотека от визуални компоненти, дизайнер на форми), наречен *Lazarus* http://lazarus.freepascal.org/ . Самият аз съм автор на *Castle Game Engine*, https://castle-engine.io/ , която е 3D и 2D библиотека с отворен код, използваща Паскал за създаване на игри на много платформи (Windows, Linux, macOS, Android, iOS, Nintendo Switch; подготвя се и WebGL).

Това въведение е насочено най-вече към програмисти, които вече имат опит в програмирането на други езици. Тук няма да разглеждаме значенията на някои универсални концепции като *"Какво е клас"*, само ще покажем как да ги използваме в Паскал.

2. Основи

2.1. Програма "Hello world"

```
// Използвайте този ред във всички нови програми
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
// Needed for console programs on Windows,
// otherwise (with Delphi) the default is GUI program without console.
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

program MyProgram; // Запишете файла като myprogram.dpr
begin
  WriteLn('Hello world!');
end.
```

Това е завършена програма, която можете да *компилирате* и *стартирате*.

- Ако използвате FPC на командния ред, просто създайте нов файл `myprogram.dpr` и изпълнете `fpc myprogram.dpr`.
- Ако използвате *Lazarus*, създайте нов проект (меню *Project → New Project → Simple Program*). Запазете го като `myProgram` и поставете по-горния код като основен файл. Компилирайте го с помощта на командата от менюто *Run → Compile*.
- Това е програма за командния ред, така че и в двата случая — просто стартирайте компилирания изпълним файл от командния ред.

В останалата част от тази книга се говори за езика Обектен Паскал, така че не очаквайте да видите нещо по-забавно от програми за команден ред. Ако искате да видите нещо по-така, просто създайте нов GUI проект в *Lazarus* (*Project → New Project → Application*). Готово — работещо GUI приложение, крос-платформено, с естествен изглед навсякъде, използвайки удобна библиотека с визуални компоненти. *Lazarus* и *Free Pascal Compiler* се предлагат с много готови модули за работа в мрежа, GUI, база данни, файлови формати (XML, JSON, изображения ...), многозадачност и всичко останало, от което може да се нуждаете. Вече споменах моя готин *Castle Game Engine* по-рано :)

2.2. Функции, процедури, примитивни типове

```
program MyProgram;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

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

function MyFunction(const S: string): string;
begin
  Result := S + 'низовете се управляват автоматично';
end;

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

  // Делението с "/" винаги дава резултат float,
  // използвайте "div" за целочислено делене
  X := 15 / 5;
  WriteLn('X сега е: ', X); // научна нотация
  WriteLn('X сега е: ', X:1:2); // 2 десетични знака
```

```
end.
```

За да върнете стойност от функция, задайте стойност на магическата променлива `Result`. Можете да четете и присвоявате свободно `Result`, точно както и всяка друга локална променлива.

```
function MyFunction(const S: string): string;
begin
  Result := S + 'нещо';
  Result := Result + ' още нещо!';
  Result := Result + ' и още!';
end;
```

Можете също да използвате и името на функцията (`MyFunction` в горния пример) като променлива, на която да присвоите резултата. Но не бих го препоръчал в нов код, тъй като изглежда "съмнително", когато се използва в дясната страна на оператор за присвояване. Просто използвайте `Result` винаги, когато искате да прочетете или да зададете резултата от функцията.

Разбира се може да го направите ако искате да извикате функцията рекурсивно. Ако извиквате рекурсивно функция без параметри, уверете се, че сте сложили скобите `()` след името (въпреки че в Паскал обикновено можете да ги пропуснете в този случай). Това ще направи рекурсивното извикване на функция без параметри различимо от прочитането на текущата стойност на функцията. Например така:

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

Можете да извикате `Exit` за да приключите изпълнението на процедурата или функцията преди тя да е достигнала последния си `end;`. Ако извикате `Exit` без параметри във функция, тогава ще се върне последното нещо присвоено на `Result`. Може да се използва и конструкцията `Exit(X)`, за да се зададе резултата от функцията и да се излезе **сега** — точно както `return X` в C-подобните езици.

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

```
end;
```

Обърнете внимание, че резултатът от функцията може да бъде игнориран. Всяка функция може да се използва и като процедура. Това има смисъл, когато функцията има някакъв *страничен ефект* (напр. променя глобална променлива) вместо да изчислява резултат. Например:

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

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

begin
  Count := 10;
  CountMe; // функцията се изпълнява но резултата й се игнорира, Count сега е 11
  MyCount := CountMe; // резултата от функцията се използва, MyCount става равно на
Count, което сега е 12
end.
```

2.3. Проверки (if)

Използвайте `if .. then` или `if .. then .. else` за да изпълните някакъв код, когато е удовлетворено определено условие. За разлика от C-подобните езици, в Паскал не е необходимо да ограждате условието в скоби.

```
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;
```

```
  // еквивалентно на горното
  B := A > 10;
  if B then
    DoSomething
  else
    DoSomethingElse;
end;
```

Клаузата `else` се отнася към последния `if`. Така че следното ще работи, както се очаква:

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

Въпреки че горния пример с вложени `if` е коректен, винаги в такива случаи е по-добре вложения `if` да се огради в `begin` ... `end` блок. Това прави кода по-очевиден за читателя и той ще остане такъв, дори ако объркате отстъпа отляво. По-долу е подобрената версия на горния пример. Когато добавите или премахнете някоя клауза `else` в долния код, винаги ще е ясно към кое условие ще бъде тя (към проверката на `A` или към проверката на `B`), така че е по-малко вероятно да се допуснат грешки.

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

2.4. Логически, релационни и побитови оператори

Логически оператори се наричат `and`, `or`, `not`, `xor`. Тяхното значение вероятно е очевидно (потърсете *"exclusive or"* ако не сте сигурни какво върши *xor*). Те вземат *boolean аргументи* и връщат *boolean* резултат. Те също могат да действат и като *побитови оператори* когато и двата аргумента са цели числа, в този случай те връщат цяло число.

Релационни (сравнителни) оператори са `=`, `<>`, `>`, `<`, `<=`, `>=`. Ако сте свикнали с C-подобни езици, обърнете внимание, че в Паскал сравнението на две стойности (проверката дали са равни), се прави като използвате само един символ на равенство `A = B` (За разлика от C, където използвате два `A == B`). Специалният оператор *assignment* в Паскал е `:=`.

Логическите (или побитовите) оператори имат по-висок приоритет от релационните оператори. Може да се наложи да използвате скоби около някои изрази, за да получите желания ред на изчисление.

Например това е грешка при компилация:

```
var
  A, B: Integer;
begin
  if A = 0 and B <> 0 then ... // НЕКОРЕКТЕН пример
```

Горното не успява да се компилира, тъй като първо компилаторът иска да изпълни побитовия `and` в средата на израза: `(0 and B)`. Това е побитова операция, която връща цяло число. След това компилатора изпълнява оператора `=`, чийто резултат е логическа стойност `A = (0 and B)`. Накрая се получава грешка *"type mismatch"* след опита да се сравни логическата стойност `A = (0 and B)` и цялото число `0`.

Това е вярно:

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

В изчислението на логически изрази се използва т.н. *кратко оценяване* (short-circuit evaluation). Разглеждаме следния израз:

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

- Гарантирано е, че първо ще бъде оценена `MyFunction(X)`.
- Ако `MyFunction(X)` върне `false`, тогава стойността на израза е известна (стойността на `false and каквото_и_да_е` е винаги `false`), и `MyOtherFunction(Y)` няма да се извика изобщо.
- Подобно е правилото и за `or` изрази. Тогава, ако израза е ясно, че е `true` (защото първия операнд е `true`), втория операнд не се оценява.
- Това е особено полезно, когато пишете изрази като

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

  Това ще сработи превилно, дори когато `A` е `nil`. Ключовата дума `nil` е за указател, равен на нула (когато е представен като число). Нарича се *null pointer* в много други езици за програмиране.

2.5. Тестване на единичен израз за множество стойности (case)

Ако трябва да се изпълни различно действие в зависимост от стойността на някакъв израз, тогава е полезна конструкцията `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;
```

Клаузата `else` е незадължителна (и съответства на `default` в C-подобните езици). Когато нито една стойност не съвпада и не е зададена `else` клауза, тогава не се изпълнява нищо.

Ако познавате C-подобни езици и сравните това с оператор `switch`, ще забележите, че няма автоматично *пропадане* (fall-through) към следващия клон. Това е умишлена благодат в Паскал. Не е нужно да помните и да поставяте инструкции `break`. При всяко изпълнение, се изпълнява *най-много един* клон на `case`, това е всичко.

2.6. Изброени и бройни типове, множества и масиви с постоянна дължина

Изброеният тип в Паскал е много удобен, непрозрачен тип. Вероятно ще го използвате много по-често от enums в другите езици:)

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

Прието е пред имената в изброения тип да се сложи двубуквен префикс от името на типа, оттук `ak` = префикс за *"Animal Kind"*. Това е полезно правило, тъй като имената на изброения тип са в глобалното пространство от имена. Така че ако им сложите префикс `ak`, вие намалявате възможността за конфликт с други идентификатори.

ЗАБЕЛЕЖКА	Конфликтите в имената не са фатални. Възможно е различните модули да дефинират един и същ идентификатор. Но е добра идея да се опитате да избягвате конфликтите така или иначе, за да поддържате кода лесен за разбиране и анализ.
ЗАБЕЛЕЖКА	Можете да избегнете дефинирането на имената от изброения тип в глобалното пространство от имена чрез компилаторната директива `{$scopedenums on}`. Това означава, че ще трябва да ги указвате винаги квалифицирани по име на тип, напр. `TAnimalKind.akDuck`. В такъв случай нуждата от префикс `ak` отпада и вероятно тогава просто ще ги наречете `Duck, Cat, Dog`. Това е подобно на C# enums.

Фактът, че изброения тип е *непрозрачен* означава, че не е възможно да се присвои директно към и от целочислен тип. Ако това е необходимо, може да се използва `Ord(MyAnimalKind)` за да се преобразува изброен тип към целочислен, или `TAnimalKind(MyInteger)` за да се преобразува целочислен тип към изброен. В последния случай първо се уверете, че `MyInteger` е в диапазона (0 .. `Ord(High(TAnimalKind))`).

Изброените и бройните типове могат да се използват за индекси на масиви:

```
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;
```

Те също могат да се използват за създаване на множества (побитови полета):

```
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. Цикли (for, while, repeat, for .. in)

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}
{$R+} // включена проверка на диапазона - подходящо за дебъг

var
  MyArray: array [0..9] of Integer;
  I: Integer;
begin
  // инизиализация
  for I := 0 to 9 do
    MyArray[I] := I * I;
```

```
  // показване
  for I := 0 to 9 do
    WriteLn('Квадрата е ', MyArray[I]);

  // прави същото като горното
  for I := Low(MyArray) to High(MyArray) do
    WriteLn('Квадрата е ', MyArray[I]);

  // прави същото като горното
  I := 0;
  while I < 10 do
  begin
    WriteLn('Квадрата е ', MyArray[I]);
    I := I + 1; // или "I += 1", или "Inc(I)"
  end;

  // прави същото като горното
  I := 0;
  repeat
    WriteLn('Квадрата е ', MyArray[I]);
    Inc(I);
  until I = 10;

  // прави същото като горното
  // забележка: тук се изброяват стойностите на MyArray, а не индексите
  for I in MyArray do
    WriteLn('Квадрата е ', I);
end.
```

Относно циклите `repeat` и `while`:

Има две разлики между тези типове цикли:

1. Условието за цикъл има противоположен смисъл. В цикъла `while .. do` условито казва *кога да се продължи*, но в `repeat .. until` условието казва *кога да се спре*.
2. При цикъла `repeat` *условието не се проверява в началото*. По този начин цикъла `repeat` винаги се изпълнява поне веднъж.

Относно цикъл `for I := ⋯`:

Цикълът `for I := .. to .. do ⋯` е близък до C-подобния цикъл `for`. Въпреки това е по-ограничен, защото не може да му се укаже произволно действие и / или произволно условие за контрол на цикъла. Той може да се изпълнява само с последователни числа (или други бройни типове). Единствената различна възможност е тази, че може да се използва `downto` вместо `to`, за да се брои наобратно.

За сметка на това той изглежда прост и изпълнението му е силно оптимизирано. По-конкретно, *изразите за горната и долната граници се изчисляват само веднъж* преди цикъла да започне.

Обърнете внимание, че стойността на променливата на брояча на цикъла (в примера `I`) се счита за *неопределена* след приключването на цикъла заради възможните оптимизации. Прочитане на стойността на `I` след цикъла може да доведе до издаване на предупреждение от компилатора. В случай обаче на предсрочно излизане с `Break` или `Exit`, променливата гарантирано запазва последната си стойност.

Относно цикъл `for I in ⋯`:

Цикълът `for I in .. do ..` е подобен на `foreach` в повечето модерни езици за програмиране. Той може да работи с много от вградените типове:

- Може да се изпълни за всички стойности в масив (горния пример).
- Може да се изпълни за всички стойности на изброен тип:

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

- Може да се изпълни за всички елементи включени в множество:

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

- И работи с потребителски типове списъци, включително генерици, като `TObjectList` or `TFPGObjectList`.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses
  SysUtils, FGL;

type
  TMyClass = class
    I, Square: Integer;
  end;
  TMyClassList = {$ifdef FPC}specialize{$endif} TFPGObjectList<TMyClass>;

var
  List: TMyClassList;
  C: TMyClass;
  I: Integer;
```

```
begin
  List := TMyClassList.Create(true); // true = притежава елементите си
  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('Квадрата на ', C.I, ' е ', C.Square);
  finally
    FreeAndNil(List);
  end;
end.
```

Все още не сме обяснили концепцията за класовете, така че последният пример може да не е съвсем очевиден. Просто продължете напред и по-късно ще стане ясно :)

2.8. Изпечатване на информация, логове

За изпечатване на низове в Паскал, използвайте процедурите `Write` или `WriteLn`. Във втората автоматично се добавя символ за нов ред накрая.

Това е "вълшебна" процедура в Паскал. Тя може да приеме променлив брой аргументи и те могат да имат почти всякакъв тип. Всички подадени аргументи се преобразуват в низове при изпечатването и има специален синтаксис за определяне ширината на полето и броя десетични цифри след запетаята.

```
WriteLn('Hello world!');
WriteLn('Може да отпечатате цяло число: ', 3 * 4);
WriteLn('Може да разширите полето на цяло число: ', 666:10);
WriteLn('Може да отпечатате число с плаваща запетая: ', Pi:1:4);
```

За да вмъкнете изрично нов ред в низа, използвайте константата `LineEnding` (от FPC RTL). (*Castle Game Engine* също така дефинира по-кратката константа `NL`.) Паскал не интерпретира никакви специални поредици в низовете, така че изписването на

```
WriteLn('One line.\nSecond line.'); // НЕКОРЕКТЕН пример
```

Не работи така, както някои от вас биха очаквали. Ще работи това:

```
WriteLn('Първи ред.' + LineEnding + 'Втори ред.');
```

или това:

```
WriteLn('Първи ред.');
WriteLn('Втори ред.');
```

Обърнете внимание, че това ще работи само в *конзолно* приложение. Уверете се, че имате дефиниция `{$apptype CONSOLE}` а **не** `{$apptype GUI}` в основния файл на програмата. В някои операционни системи няма значение и винаги ще работи (Unix), но на други (Windows) опита за изпечатване с `Write` или `WriteLn` в GUI приложение ще предизвика грешка.

В **Castle Game Engine:** използвайте `WriteLnLog` или `WriteLnWarning` вместо `WriteLn` за печат на диагностична информация. Те винаги ще бъдат насочени към някакво полезно устройство или файл. В Unix това ще бъде стандартния изход. В Windows GUI приложение ще бъде лог-файл. В Android ще бъде *Android logging facility* (може да се прочете с `adb logcat`). Използването на `WriteLn` трябва да се ограничи до случаите, в които се пишат конзолни приложения (например 3D моделен конвертор / генератор) и знаете, че *стандартния изход* съществува.

2.9. Преобразуване в низ

За конвертиране на произволен брой аргументи в низ (вместо просто директно да ги извеждате) съществуват няколко възможности.

- Може да конвертирате определени типове в низ като използвате специализираните функции като `IntToStr` и `FloatToStr`. Освен това, в Паскал да можете да конкатенирате (свързвате) низове просто като използвате оператора за събиране. По този начин можете да създадете низ подобен на следния: `'Моето цяло число е ' + IntToStr(MyInt) + ' и стойността на Pi е ' + FloatToStr(Pi)`.
  - *Предимство*: Изключително удобно. Съществува множество готови функции `XxxToStr` и подобни на тях (например `FormatFloat`), покриващи много типове. Повечето от тях са в модула `SysUtils`.
  - *Друго предимство*: Почти винаги съществува и обратна функция. За конвертиране на низ (напр. въведен от потребителя) обратно до цяло число или до число с плаваща запетая, може да се използват `StrToInt`, `StrToFloat` и подобни на тях (например `StrToIntDef`).
  - *Недостатък*: Дълга конкатенация от много извиквания на `XxxToStr` и низове не изглежда красиво.
- Функцията `Format`, използва се по следния начин: `Format('%d %f %s', [MyInt, MyFloat, MyString])`. Тя е подобна на функцията `sprintf` в C-подобните езици. Тя вмъква аргументите си на зададените места в указания шаблон. Така зададените места може да използват специален синтаксис за уточняване на формата, напр. `%.4f` означава число с плаваща запетая с 4 знака след запетаята.
  - *Предимство*: Разделянето на *шаблона* от *аргументите* изглежда по-чисто и спретнато. Ако искате да промените *шаблона* без да закачате аргументите (напр. при езиков превод), лесно може да го направите.

    - *Друго предимство*: Няма никаква компилаторна магия. Може да използвате същия синтаксис за да подадете всякакъв брой аргументи от произволен тип в собствените си подпрограми (декларирайте параметър като `array of const`). След това можете да предадете тези аргументи надолу към `Format`, или да разчлените листа с аргументи и да правите каквото си искате с тях.
    - *Недостатък*: Компилатора не проверява дали шаблона съвпада с броя и типа на аргументите. Използването на неподходящ синтаксис на конкретното място в шаблона ще предизвика изключение по време на изпълнение (`EConvertError` а не нещо гадно като грешка в сегментацията).
- `WriteStr(TargetString, ⋯)` процедурата работи по същия начин както `Write(⋯)`, с изключение на това, че резултата се записва в `TargetString` вместо да се отпечати.
    - *Предимство*: Поддържа всички функционалности на `Write`, включително специалния синтаксис за форматиране за ширина на полето и знаци след запетаята, напр. `Pi:1:4`.
    - *Недостатък*: Синтаксиса за форматиране е като "компилаторна магия", направена конкретно за процедури като тази. Това понякога е проблем, защото не можете да направите собствена процедура `MyStringFormatter(⋯)`, която да позволява използването на нещо подобно на `Pi:1:4`. Поради тази причина (и защото дълго време не е била имплементирана в основните Паскал компилатори), конструкцията не е много популярна.

3. Модули (Unit-и)

Unit-ите позволяват групиране на общи елементи (всички, които могат да се декларират), за използване от други unit-и и програми. Те са еквиваленти на *модулите* и *пакетите* в други езици за програмиране. Имат секция *interface*, където се декларират елементите достъпни за използване от другите unit-и и програми и секция *implementation* където е описано как тези елементи работят. Може да запишете unit-а `MyUnit` под името `myunit.pas` (малки букви с разширение `.pas`).

```
unit MyUnit;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}

interface

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

implementation

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

function MyFunction(const S: string): string;
```

```
begin
  Result := S + 'низовете се управляват автоматично';
end;

end.
```

Основната програма се записва обикновено под име `myprogram.lpr` (`lpr` = Lazarus program file; в Delphi обикновено се използва `.dpr`). Трябва да се спомене, че са възможни и други разширения, някои проекти използват `.pas` за основната програма, някои използват `.pp` за unit-и или програми. Аз препоръчвам използването на `.pas` за unit-и и `.lpr` за FPC/Lazarus програми.

Програма може да използва unit със служебната дума `uses`:

```
program MyProgram;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses MyUnit;

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

Unit-а може да съдържа секции `initialization` и `finalization`. Кода в тези секции се изпълнява когато програмата стартира или респективно — приключва.

```
unit initialization_finalization;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}

interface

implementation

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

3.1. Unit-и, които се използват взаимно

Един unit може да използва друг unit. Другия unit може да се използва в секцията interface или само в секцията implementation. Първото позволява да се дефинират нови публикувани

елементи (процедури, типове,...) на базата на вече известните от другия unit. Второто е по-ограничено, т.е. ако използвате unit само в секцията implementation, неговите идентификатори важат само в нея.

```
unit AnotherUnit;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}

interface

uses Classes;

{ Типът (клас) "TComponent" е дефиниран в unit Classes.
  Поради тази причина трябва да използваме uses Classes; по-горе. }
procedure DoSomethingWithComponent(var C: TComponent);

implementation

uses SysUtils;

procedure DoSomethingWithComponent(var C: TComponent);
begin
  { Процедурата FreeAndNil е дефинирана в unit SysUtils.
    Тъй като го използваме само в реализацията а не в интерфейсната част,
    достатъчно е да използваме uses SysUtils; в секция "implementation". }
  FreeAndNil(C);
end;

end.
```

Не е позволено да има *кръгови зависимости между unit-и* в техния интерфейс. Това означава, че не може два unit-а да се използват взаимно в секцията interface. Причината за това е, че за да "разбере" интерфейсната част на даден unit, компилатора трябва първо да "разбере" интерфейсната част на всички други unit-и, които той използва. Езикът Паскал спазва това правило много стриктно и това позволява бързата компилация и автоматичното определяне *какво е нужно да се прекомпилира*. Няма необходимост да се използват сложни файлове `Makefile` за простата задача по компилирането, както и също няма нужда от *прекомпилиране на всичко* само за да се уверим, че всички зависимости са се обновили правилно.

Напълно е възможно кръговото използване на unit-и при условие, че *поне единият от тях се използва в секция implementation*. Така например unit `A` може да използва `B` в секцията си interface а от друга страна unit `B` може да използва unit `A` в секцията си implementation.

3.2. Квалифициране на идентификаторите с името на unit-а

Различни unit-и може да дефинират един и същи идентификатор. За да бъде кода прост за четене и търсене, това би трябвало да се избягва но не винаги е възможно. В тези случаи обикновено "печели" последния включен unit в клаузата `uses`, което означава че неговите идентификатори скриват тези със същите имена от предишните unit-и.

Винаги може да укажете изрично unit-а за даден идентификатор като използвате името на unit-а пред него разделено с точка `MyUnit.MyIdentifier`. Това е стандартното решение за ситуации, в които желания идентификатор от `MyUnit` е скрит от друг unit. Разбира се може също да промените реда на unit-ите в клаузата uses, но пък това ще засегне и всички други дефинирани идентификатори.

```
program showcolor;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

// И двата unit-а Graphics и GoogleMapsEngine дефинират тип TColor.
uses Graphics, GoogleMapsEngine;

var
  { Това не работи както ни се иска, оказва се, че TColor е
    дефиниран от GoogleMapsEngine. }
  // Color: TColor;
  { Това работи. }
  Color: Graphics.TColor;
begin
  Color := clYellow;
  WriteLn(Red(Color), ' ', Green(Color), ' ', Blue(Color));
end.
```

За unit-ите трябва да се запомни, че имат две `uses` клаузи: едната в част interface и другата в част implementation. Правилото *следващите unit-и скриват идентификаторите на предишните* се прилага навсякъде, което означава и че *unit-ите използвани в част implementation* могат да скрият идентификатори от unit-и *използвани в секция interface*. От друга страна, факта че за секция `interface` имат значение само unit-ите използвани в interface, може да доведе до объркващи ситуации, в които привидно еднакви декларации се приемат за различни от компилатора:

```
unit UnitUsingColors;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}

// НЕКОРЕКТЕН пример
```

```
interface

uses Graphics;

procedure ShowColor(const Color: TColor);

implementation

uses GoogleMapsEngine;

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

end.
```

В unit `Graphics` (от Lazarus LCL) се дефинира тип `TColor`. Но компилатора няма да компилира горния unit, твърдейки че не сте написали тяло на процедурата `ShowColor`, която да отговаря на декларацията в interface. Проблемът е че unit `GoogleMapsEngine` също дефинира тип с името `TColor`. Понеже се използва само в секция `implementation`, тази дефиниция *засенчва* дефиницията `TColor` само в implementation. Еквивалентната версия на горния unit, където грешката е очевидна, би изглеждала така:

```
unit UnitUsingColors;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}

// НЕКОРЕКТЕН пример
// Ето какво "вижда" компилатора когато се опитва да компилира предишното

interface

uses Graphics;

procedure ShowColor(const Color: Graphics.TColor);

implementation

uses GoogleMapsEngine;

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

end.
```

Решението на проблема в случая е просто — укажете изрично в implementaton да се

използва TColor от unit Graphics. Може и да го оправите като преместите GoogleMapsEngine в секция interface преди Graphics. Това обаче ще доведе до други последици в unit-а UnitUsingColors защото ще се отрази на всичките му дефиниции.

```
unit UnitUsingColors;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}

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. Представяне на идентификаторите от един unit чрез друг

Понякога искате да вземете идентификатор от един unit и да го *представите* чрез друг. Крайният резултат трябва да бъде, че когато използвате новия unit, стария идентификатор ще бъде достъпен в пространството на имената.

Понякога това е необходимо за да се запази съвместимостта с по-стари версии на unit-а. А понякога е удобно да се "скрие" някой unit само за вътрешно ползване.

Това може да се направи с повторна дефиниция на идентификатора в новия unit.

```
unit MyUnit;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}

interface

uses Graphics;

type
  { Представи TColor от unit Graphics като TMyColor. }
  TMyColor = TColor;
```

```
  { Алтернативно, представи го под същото име.
    Квалифицирай типа с името на unit-а, в противен случай ще изглежда,
    че типа се позовава сам на себе си "TColor = TColor" в дефиницията. }
  TColor = Graphics.TColor;

const
  { Може така да представите и константи от друг unit. }
  clYellow = Graphics.clYellow;
  clBlue = Graphics.clBlue;

implementation

end.
```

Трябва да се отбележи, че този трик не може така лесно да се направи с глобалните процедури, функции и променливи. С процедурите и функциите можете да обявите константен указател към процедура в друг unit (виж Callbacks (познати като *Събития*, също като *Указатели към функции*, също като *Процедурни променливи*)), но това изглежда доста "нечисто".

Обикновено решението се състои в създаване на "опаковъчни" функции[4], които просто извикват старите от другия unit, като им подават параметрите и връщат резултата.

За да се направи нещо подобно с глобалните променливи, може да се използват глобални свойства (unit-level properties), виж Свойства.

4. Класове

4.1. Основи

В Паскал се използват класове (classes). На базово ниво класовете са просто контейнери за:

- *полета (fields)* (друго име за *"променлива вътре в класа"*),
- *методи (methods)* (друго име за *"процедура или функция вътре в класа"*),
- и *свойства (properties)* (удобен синтаксис за нещо, което е подобно на поле, но всъщност е двойка методи за *четене (get)* и *запис (set)* на някаква стойност; повече за това в Свойства).
- Общо казано, в един клас може да се вместят много други неща, повече е описано в Допълнителни декларации и вложени класове.

```
type
  TMyClass = class
    MyInt: Integer; // това е поле
    property MyIntProperty: Integer read MyInt write MyInt; // това е свойство
    procedure MyMethod; // това е метод
```

```
  end;

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

4.2. Наследяване, проверка (is), конверсия на типа (as)

Паскал поддържа *наследяване* на класове и *виртуални* методи.

```
program MyProgram;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses
  SysUtils;

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

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

procedure TMyClass.MyVirtualMethod;
begin
  WriteLn('TMyClass shows MyInt + 10: ', MyInt + 10);
end;

procedure TMyClassDescendant.MyVirtualMethod;
begin
  WriteLn('TMyClassDescendant shows 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 подразбиране методите не са виртуални, за да бъдат такива трябва да се декларират със запазената дума `virtual`. Подмяната на виртуален метод трябва да се декларира с `override`, в противен случай ще се изведе предупреждение. За да скриете метод без да го подменяте трябва да се използва думата `reintroduce` (обикновено това се прави само ако имате основателна причина).

За да се провери какъв е класа на обектна инстанция по време на изпълнение се използва оператора `is`. За да се смени типа на инстанция, т.е. да се конвертира до друг клас, се използва оператора `as`.

```
program is_as;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses SysUtils;

type
  TMyClass = class
    procedure MyMethod;
  end;

  TMyClassDescendant = class(TMyClass)
    procedure MyMethodInDescendant;
  end;

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

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

var
  Descendant: TMyClassDescendant;
  C: TMyClass;
```

```
begin
  Descendant := TMyClassDescendant.Create;
  try
    Descendant.MyMethod;
    Descendant.MyMethodInDescendant;

    { Descendant има цялата функционалност, която се очаква от
      TMyClass, така че това присвояване е OK }
    C := Descendant;
    C.MyMethod;

    { Това не може да сработи, тъй като TMyClass не дефинира този метод }
    //C.MyMethodInDescendant;
    if C is TMyClassDescendant then
      (C as TMyClassDescendant).MyMethodInDescendant;

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

Вместо `X as TMyClass`, може да използвате и конвертиране *без проверка* `TMyClass(X)`. Това е по-бързо от предишното, но резултата може да доведе до неопределено поведение ако `X` не се явява наследник на `TMyClass`. Поради тази причина не използвайте `TMyClass(X)`, освен ако не е **абсолютно** сигурно, че `X` е наследник на `TMyClass`, например ако преди това сте проверили с `is`:

```
if A is TMyClass then
  (A as TMyClass).CallSomeMethodOfMyClass;
// долното е малко по-бързо
if A is TMyClass then
  TMyClass(A).CallSomeMethodOfMyClass;
```

4.3. Свойства

Свойствата са много удобна *"синтактична захар"* (б.пр. syntax sugar - особеност на синтаксиса, която не влияе на поведението на програмата, но прави използването на езика по-удобно) за:

1. Нещо да изглежда като поле (да може да се чете и записва) но под него да има методи за четене (*getter*) и запис (*setter*). Често се използва за получаване на странични ефекти (напр. обновяване на екрана) всеки път когато стойността се промени;
2. Нещо да изглежда като поле, но да е само за четене. В резултат на това полето е подобно на константа или функция без аргументи.

```
type
  TWebPage = class
```

```
  private
    FURL: string;
    FColor: TColor;
    function SetColor(const Value: TColor);
  public
    { Няма начин да се запише директно.
      Извикайте метода Load, например Load('http://www.freepascal.org/'),
      за да заредите страницатата и да установите свойството. }
    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;
    // за пример: предизвиква обновяване всеки път при промяна на стойността
    Repaint;
    // пак за пример: осигурява, че някаква друга вътрешна инстанция,
    // като "RenderingComponent" (каквато и да е тя),
    // съдържа същата стойност за Color.
    RenderingComponent.Color := Value;
  end;
end;
```

Забележете, че вместо да се укаже метод, може да се укаже и име на поле (обикновено частно поле) за директно четене или запис. В горния пример, свойството `Color` използва метод за запис (*setter* `SetColor`. Но за прочитане на стойността свойството `Color` указва директно към частното поле `FColor`. Указването на поле е по-бързо отколкото извикването на "опъковъчен" метод за четене или запис. По-бързо е както за писане, така и за изпълнение.

Когато се декларира свойство трябва да се укаже:

1. Дали може да се чете и как (с директен достъп до поле или с извикване на метод `getter`);
2. И съответно — дали може да се записва и как (с директен достъп до поле или с използване на метод `setter`).

Компилатора проверява дали типовете на указаните полета и методи съответстват на типа на свойството. Например, за да прочетете `Integer` свойство, трябва да укажете или поле от тип `Integer` или метод без параметри, който връща `Integer`.

Технически, за компилатора методите "getter" и "setter" са просто нормални методи и могат да правят абсолютно всичко (включително странични ефекти или рандомизация). Но е добра практика свойствата да се проектират така, че да се държат повече или по-малко като полета:

- Функцията *getter* не би трябвало да има видими странични ефекти (напр. не трябва да чете от файл или от клавиатурата). Четенето трябва да е детерминистично (без рандомизация, дори псевдо-рандомизация :). Многократното четене на свойство трябва да връща една и съща стойност ако нищо не се е променило междувременно.

  Напълно в реда на нещата е *getter* да има някакви *невидими* странични ефекти, например да съхрани стойностите от някакво изчисление за да се ускори изпълнението при следващо извикване. Това е една от полезните функции на методите "getter".

- Функцията *setter* трябва винаги да запише подадената стойност, по такъв начин, че извикването на *getter* да я върне обратно. Не бива некоректните стойности автоматично да се игнорират в "setter", в такива случаи е редно да се предизвика изключение (exception). Не е добре също стойността да се конвертира или мащабира. Идеята е, че след `MyClass.MyProperty := 123;` програмиста трябва да очаква, че `MyClass.MyProperty = 123`.
- Свойствата само за четене, *read-only properties*, често се използват за да е възможно само четенето на някое поле отвън. Отново, добрата практика е това свойство да се държи като константа или поне като константа за текущото състояние на обекта. Стойността не бива да се променя неочаквано. *Ако четенето предизвиква странични ефекти или се връща случайна стойност, вместо свойство трябва да се използва функция.*
- Полето, към което се обръща свойството трябва винаги да е *private* защото идеята на свойството е да "капсулира" целия външен достъп до него.
- Технически е възможно да се направи свойство само за запис, *set-only property*, но още не съм видял добър пример за какво може да послужи такова свойство :)

ЗАБЕЛЕЖКА	Свойствата могат да се дефинират и извън клас, на ниво unit. Такива свойства служат за аналогични цели — изглеждат като глобални променливи, но четенето и записа им извиква указаните подпрограми за *getter* и *setter*.

4.3.1. Сериализация на свойства

Публикуваните свойства са база за *сериализацията* (или *streaming components*) в Паскал. *Сериализация* означава, че данните на инстанцията от даден клас се записват в поток (*stream*, подобно на файл), от който може по-късно да се прочетат обратно.

Сериализирането е това, което се случва, когато Lazarus чете (или записва) състоянието на компонент във файл `xxx.lfm`. (В Delphi еквивалентния файл има разширение `.dfm`). Този механизъм може да се използва и за други цели с помощта на процедури като `ReadComponentFromTextStream` от unit `LResources`. Също така може да се използват и други сериализационни алгоритми, например от unit `FpJsonRtti` (сериализация в JSON формат).

В **Castle Game Engine**: Използвайте unit `CastleComponentSerialize` (базиран на `FpJsonRtti`) за да

сериализирате нашите компоненти като user-interface и transformation component hierarchies.

За всяко свойство може да се декларират допълнителни полезни неща за алгоритъма за сериализация:

- Може да укажете подразбираща се стойност за свойството (с резервираната дума `default`). Обърнете внимание, че така или иначе в конструктора е необходимо да се инициализира това свойство с тази конкретна стойност по подразбиране. Това не се прави автоматично. Декларацията `default` е само информативна за сериализиращия алгоритъм: *"когато конструктора се изпълни, даденото свойство има дадената стойност"*.
- Дали свойството трябва да се записва изобщо (с резервираната дума `stored`).

4.4. Изключения - Кратък пример

В Паскал може да се предизвикват и обработват изключения. Обработката се прави с клаузи `try ⋯ except ⋯ end`, също така има и финални секции `try ⋯ finally ⋯ end`.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

program MyProgram;

uses
  SysUtils;

type
  TMyClass = class
    procedure MyMethod;
  end;

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

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

```
end.
```

Обърнете внимание, че клаузата `finally` се изпълнява дори ако излезете от блок с използването на `Exit` (от функция / процедура / метод) или `Break` или `Continue` (от тялото на цикъл).

Виж глава Изключения за по-задълбочено описание на *изключенията*.

4.5. Нива на видимост

Както в повечето обектно-ориентирани езици, в Паскал има спецификатори за ограничаване на видимостта на полета / методи / свойства.

Основните нива на видимост са:

`public`
: всеки може да го достъпи, в това число и кода от други unit-и.

`private`
: достъпно само в този клас.

`protected`
: достъпно само в този клас и наследниците му.

Даденото по-горе обяснение за `private` и `protected` не е напълно вярно. Кодът *в същия unit* може да прескача ограничението и да достъпва неща, които са указани като `private` или `protected`. Понякога това е удобно, тъй като позволява създаване на по-силно свързани класове. Използвайте `strict private` или `strict protected` за да обезопасите вашите класове още по-стриктно. По-подробно това е описано в Частни и лични полета.

Ако не укажете видимост, по подразбиране се приема `public`. Изключение се прави за класовете компилирани с директивата `{$M+}`, или наследници на класове компилирани с `{$M+}`, което включва всички наследници на `TPersistent`, също така включва и всички наследници на `TComponent` (защото `TComponent` е наследник на `TPersistent`). За тях видимостта по подразбиране е `published`, което е като `public`, но с допълнението, че системата за сериализация знае как да ги обработва.

Не всяко поле и свойство може да бъде в секция `published` (не веки тип може да се сериализира и само класове от прости полета могат да се сериализират). Просто използвайте `public`, ако не ви е грижа за сериализацията, но искате нещо да е достъпно за всички ползватели.

4.6. Предшественик по подразбиране

Ако не декларирате предшестващ клас, то по подразбиране се приема, че се наследява класа `TObject`.

4.7. Self

Резервираната дума `Self` (аз) може да се използва в реализацията на класа за да укаже изрично, че става дума за вашата собствена инстанция. Това е еквивалент на `this` от C++, Java и подобни езици.

4.8. Извикване на наследен метод

В рамките на реализация на метод, ако извикате друг метод, тогава по подразбиране вие извиквате метода на вашия собствен клас. В примерния код по-долу, `TMyClass2.MyOtherMethod` извиква `MyMethod`, който в крайна сметка извиква `TMyClass2.MyMethod`.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses SysUtils;

type
  TMyClass1 = class
    procedure MyMethod;
  end;

  TMyClass2 = class(TMyClass1)
    procedure MyMethod;
    procedure MyOtherMethod;
  end;

procedure TMyClass1.MyMethod;
begin
  Writeln('TMyClass1.MyMethod');
end;

procedure TMyClass2.MyMethod;
begin
  Writeln('TMyClass2.MyMethod');
end;

procedure TMyClass2.MyOtherMethod;
begin
  MyMethod; // this calls TMyClass2.MyMethod
end;

var
  C: TMyClass2;
begin
  C := TMyClass2.Create;
  try
    C.MyOtherMethod;
  finally FreeAndNil(C) end;
```

```
end.
```

Ако метода не е дефиниран за дадения клас, тогава се извиква метод от предшестващия клас. Всъщност, когато извикате `MyMethod` на инстация от `TMyClass2`, тогава

- Компилатора търси `TMyClass2.MyMethod`.
- Ако не го намери, търси `TMyClass1.MyMethod`.
- Ако не го намери, търси `TObject.MyMethod`.
- Ако не го намери, дава грешка при компилация.

Може да го проверите като сложите коментар пред дефиницията на `TMyClass2.MyMethod` в по-горния пример. Като резултат от извикването на `TMyClass2.MyOtherMethod` ще се извика `TMyClass1.MyMethod`.

Понякога не искате да извиквате метода на собствения си клас а искате да извикате метода на предшественик (или предшественик на предшественик и т.н). За да направите това, добавете ключовата дума `inherited` преди извикването на `MyMethod` по следния начин:

```
inherited MyMethod;
```

По този начин вие *насилвате* компилаторът да започне да търси от предшестващия клас. В нашия пример това означава, че компилаторът търси `MyMethod` в `TMyClass1.MyMethod`, след това `TObject.MyMethod` и след това се отказва. Дори и не обмисля използването на `TMyClass2.MyMethod`.

ПОДСКАЗКА	Променете `TMyClass2.MyOtherMethod` така, че да използва `inherited MyMethod` и вижте каква ще е разликата в резултата.

Най-често извикването на наследен метод се използва от метода със същото име в наследника. По този начин наследника може да допълни и подобри предшественика запазвайки неговата функционалност вместо да я подмени изцяло. Както в примера по-долу.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses SysUtils;

type
  TMyClass1 = class
    constructor Create;
    procedure MyMethod(const A: Integer);
  end;

  TMyClass2 = class(TMyClass1)
    constructor Create;
```

```
    procedure MyMethod(const A: Integer);
  end;

constructor TMyClass1.Create;
begin
  inherited Create; // this calls TObject.Create
  Writeln('TMyClass1.Create');
end;

procedure TMyClass1.MyMethod(const A: Integer);
begin
  Writeln('TMyClass1.MyMethod ', A);
end;

constructor TMyClass2.Create;
begin
  inherited Create; // this calls TMyClass1.Create
  Writeln('TMyClass2.Create');
end;

procedure TMyClass2.MyMethod(const A: Integer);
begin
  inherited MyMethod(A); // this calls TMyClass1.MyMethod
  Writeln('TMyClass2.MyMethod ', A);
end;

var
  C: TMyClass2;
begin
  C := TMyClass2.Create;
  try
    C.MyMethod(123);
  finally FreeAndNil(C) end;
end.
```

Понеже използването на `inherited` за извикване на метод със същото име и аргументи се среща много често, има специален съкратен вариант: може да напишете само `inherited;` (ключовата дума `inherited`, следвана непосредствено от точка и запетая, вместо името на метод). Това означава "*извикай наследения метод със същото име, предавайки му същите параметри както на текущия метод*".

ПОДСКАЗКА	В горния пример, всички извиквания на `inherited ⋯;` могат да се заменят просто с `inherited;`.

Бележка 1: Този `inherited;` е наистина съкращение на извикването на наследения метод със *същите параметри*. Ако вече сте променили стойностите на параметрите (което е напълно възможно ако не са `const`), тогава наследения метод може да получи различни входни стойности от вашия наследник. Разгледайте следното:

```
procedure TMyClass2.MyMethod(A: Integer);
begin
  Writeln('TMyClass2.MyMethod начално ', A);
  A := 456;
  { Това извиква TMyClass1.MyMethod with A = 456,
    независимо от стойността на A подадена на този метод (TMyClass2.MyMethod). }
  inherited;
  Writeln('TMyClass2.MyMethod крайно ', A);
end;
```

Бележка 2: Когато много класове дефинират метода `MyMethod` (по "*веригата на наследяване*") обикновено той се прави *виртуален*. Повече за виртуалните методи има в раздела по-долу. Но ключовата дума `inherited` работи независимо дали методът е виртуален или не. `inherited` винаги означава, че компилаторът започва да търси метода в предшественика и има смисъл както за *виртуални*, така и за *не виртуални* методи.

4.9. Виртуални методи, подмяна и скриване

По подразбиране методите *не са виртуални*. Това е както в езика C++ и за разлика от Java.

Когато методът *не е виртуален*, компилаторът определя кой метод да се извика въз основа на текущия *деклариран* тип клас, а не въз основа на *действително създадения* тип клас. Разликата изглежда незначителна, но е важно, когато променливата ви е декларирана, че е от клас `TFruit`, но всъщност може да е от клас-наследник например `TApple`.

Идеята на обектно-ориентираното програмиране е, че *класът-наследник винаги е добър поне колкото наследения*, така че компилатора позволява използването на наследник винаги когато се очаква някой от предшествениците му. Когато един метод *не е виртуален*, това може да доведе до нежелани последици. Разгледайте следния случай:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses SysUtils;

type
  TFruit = class
    procedure Eat;
  end;

  TApple = class(TFruit)
    procedure Eat;
  end;

procedure TFruit.Eat;
begin
  Writeln('Изядохме плод');
end;
```

```
procedure TApple.Eat;
begin
  Writeln('Изядохме ябълка');
end;

procedure DoSomethingWithAFruit(const Fruit: TFruit);
begin
  Writeln('Имаме плод от клас ', Fruit.ClassName);
  Writeln('Ядем го:');
  Fruit.Eat;
end;

var
  Apple: TApple; // Забележка: тук също така може да декларирате "Apple: TFruit"
begin
  Apple := TApple.Create;
  try
    DoSomethingWithAFruit(Apple);
  finally FreeAndNil(Apple) end;
end.
```

Този пример ще отпечата

```
Имаме плод от клас TApple
Ядем го:
Изядохме плод
```

Всъщност извикването `Fruit.Eat` извиква имплементацията на `TFruit.Eat` и нищо не извика имплементацията на `TApple.Eat`.

Ако се замислите как работи компилатора, това ще ви се стори естествено: когато написахте `Fruit.Eat`, променливата `Fruit` бе декларирана от тип `TFruit`. Компилаторът търси метод наречен `Eat` в класа `TFruit`. Ако класът `TFruit` не съдържа такъв метод, компилаторът ще търси в предшественика (`TObject` в този случай). Но компилаторът *не може да търси в наследници (като `TApple`)*, тъй като не знае дали *действителният клас* на `Fruit` е `TApple`, `TFruit` или някакъв друг наследник на `TFruit` (като `TOrange`, не е показан в примера по-горе).

С други думи, *методът, който ще бъде извикан*, се определя по време на компилиране.

Използването на *виртуалните методи* променя това поведение. **Ако методът `Eat` е виртуален** (пример за него е показан по-долу), тогава действителната метод, която ще бъде извикан, се определя *по време на изпълнение*. Ако променливата `Fruit` съдържа екземпляр на класа `TApple` (дори ако променливата е декларирана като `TFruit`), тогава методът `Eat` ще бъде потърсен първо в `TApple`.

В Обектния Паскал, за да дефинирате метод като *виртуален*, трябва да:

- Маркирайте първата му дефиниция (в най-горния предшественик) с ключовата дума

`virtual`.

- Маркирайте всички останали дефиниции (в наследниците) с ключовата дума `override`. Всички подменени версии трябва да имат абсолютно еднакви параметри (и да връщат едни и същи типове, в случая на функции).

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses SysUtils;

type
  TFruit = class
    procedure Eat; virtual;
  end;

  TApple = class(TFruit)
    procedure Eat; override;
  end;

procedure TFruit.Eat;
begin
  Writeln('Изядохме плод');
end;

procedure TApple.Eat;
begin
  Writeln('Изядохме ябълка');
end;

procedure DoSomethingWithAFruit(const Fruit: TFruit);
begin
  Writeln('Имаме плод от клас ', Fruit.ClassName);
  Writeln('Ядем го:');
  Fruit.Eat;
end;

var
  Apple: TApple; // Забележка: тук също така може да декларирате "Apple: TFruit"
begin
  Apple := TApple.Create;
  try
    DoSomethingWithAFruit(Apple);
  finally FreeAndNil(Apple) end;
end.
```

Този пример ще отпечата

```
Имаме плод от клас TApple
Ядем го:
```

```
Изядохме ябълка
```

Вътрешно виртуалните методи работят, като използват така наречената *виртуална таблица с методи (VMT)*, свързана с всеки клас. Тази таблица е списък с указатели към виртуалните методи за този клас. Когато извиква метода `Eat`, компилаторът разглежда таблицата с виртуални методи, свързана с *действителния* клас на `Fruit`, и използва указател към конкретния метод `Eat` съхранен там.

Ако не използвате ключовата дума `override`, компилаторът ще ви предупреди, че *скривате* виртуалния метод на предшественика с невиртуална дефиниция. Ако сте сигурни, че точно това искате да направите, можете да добавите ключова дума `reintroduce`. Но в повечето случаи е по-добре да запазите метода виртуален и да добавите ключовата дума `override`, като по този начин сте сигурни, че се извиква правилно.

5. Освобождаване на паметта за класове

5.1. Помнете да освобождавате паметта заета от инстанциите

Инстанциите на класове трябва да се освобождават ръчно. В противен случай ще се получи *изтичане на памет*. Съветвам да се използват опциите `-gl -gh` на FPC за засичане на изтичания (виж https://castle-engine.io/manual_optimization.php#section_memory).

Забележете, че това не касае възникналите изключения. Въпреки че изрично създавате инстанция от клас, когато предизвиквате изключение (и това е напълно нормален клас и можете да създадете свои собствени класове за тази цел), то тази инстанция ще бъде освободена автоматично от вградения механизъм за обработка на изключения.

5.2. Как да освободим паметта

За да освободите заетата памет от инстанция на клас, най-добре извикайте `FreeAndNil(A)` от unit `SysUtils` върху нея. Тази процедура ще провери дали `A` е `nil`, ако не е — ще извика нейния деструктор (destructor) и ще и присвои стойност `nil`. Така многократното и извикване няма да доведе до грешка.

Приблизително това съответства на следното:

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

Всъщност това е доста опростено, тъй като `FreeAndNil` използва трик за да присвои `nil` на `A` **преди** да извика деструктора с подходяща препратка. Това предотвратява определен клас

грешки — идеята е, че "външният" код никога не бива да има достъп до полуразрушено копие на инстанция от класа.

Често ще видите и да се използва метода `A.Free`, което е същото като:

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

което унищожава инстанцията `A` (и освобождава заетата памет от нея) , освен ако тя е `nil`.

Забележете, че при нормални обстоятелства никога не бива да се извиква метод на инстанция, която може да е `nil`. Затова извикването `A.Free` може да изглежда подозрително на пръв поглед. Методът `Free` обаче е изключение от това правило. Той прави нещо "нечисто" в тялото си — именно проверява дали `Self <> nil`. Този трик работи само при методи, които не са виртуални (които не извикват други виртуални методи и не достъпват никакви полета).

Съветвам ви да използвате `FreeAndNil(A)` винаги, без изключения и никога да не извиквате директно метода `Free` или деструктора `Destroy`. *Castle Game Engine* работи по този начин. Това позволява да бъдете уверени в това, че *всички препратки са или* `nil`*, или сочат към валидни инстанции*.

5.3. Ръчно и автоматично освобождаване

В много случаи необходимостта от освобождаване на инстанцията не е голям проблем. Вие просто пишете деструктор, който съответства на конструктора и освобождава всичко, за което е заделена памет в конструктора (или по-точно - през целия живот на инстанцията). Внимавайте да освободите всяко нещо само **веднъж**. Добра идея е да установите освободения указател на `nil`, обикновено е най-удобно да го направите, като извикате `FreeAndNil(A)`.

Пример:

```
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;
```

За да избегнете необходимостта от изрично освобождаване, можете също да използвате функцията за *"собственост"* на `TComponent`. Обект, който е *нечий* ще бъде автоматично освободен от *собственика*. Механизмът е достатъчно съобразителен и никога няма да освободи вече освободена инстанция (така че нещата ще работят правилно, дори ако ръчно освободите притежавания обект по-рано). Можем да променим предишния пример така:

```
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;
```

Обърнете внимание, че тук трябва да заменим виртуалния конструктор на `TComponent`. Така че не можем да променим параметрите на конструктора. (Всъщност можете — декларирайте нов конструктор с `reintroduce`. Но бъдете внимателни, тъй като някои функции, например тези за сериализация, все още ще използват виртуалния конструктор, така че се уверете, че работи правилно и в двата случая.)

Имайте предвид, че винаги можете да използвате за собственик `nil`. По този начин механизмът за *"собственост"* няма да се използва за този компонент. Това има смисъл, ако трябва да използвате наследника на `TComponent`, но искате винаги да го освобождавате ръчно. За да направите това, трябва да конструирате наследника по този начин: `ManualGun := TGun.Create(nil);`.

Друг механизъм за автоматично освобождаване е функционалността `OwnsObjects` (по

подразбиране е вече `true`!) на класовете-контейнери като `TFPGObjectList` или `TObjectList`. Така че можем също да напишем:

```
uses SysUtils, Classes, FGL;

type
  TGun = class
  end;

  TGunList = {$ifdef FPC}specialize{$endif} TFPGObjectList<TGun>;

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

constructor TPlayer.Create;
begin
  inherited;
  // Всъщност, стойността true (за OwnsObjects) е зададена по подразбиране
  Guns := TGunList.Create(true);
  Gun1 := TGun.Create;
  Guns.Add(Gun1);
  Gun2 := TGun.Create;
  Guns.Add(Gun2);
end;

destructor TPlayer.Destroy;
begin
  { Трябва да се погрижим за освобождаването на списъка.
    Той ще освободи елементите си автоматично. }
  FreeAndNil(Guns);

  { Вече няма нужда да освобождаваме ръчно Gun1, Gun2. Хубав навик е да установим на
"nil"
    техните препратки, тъй като знаем, че са освободени. В този прост клас и с
    този прост деструктор, очевидно е, че те няма да бъдат достъпвани повече --
    но правейки така ще ни помогне в случая на по-големи и по-сложни деструктори.

    Алтернативно, можем да си спестим декларирането на Gun1 и Gun2,
    и вместо това да използваме Guns[0] и Guns[1] в нашия код.
    Или да създадем метод Gun1, който връща Guns[0]. }
  Gun1 := nil;
  Gun2 := nil;
  inherited;
end;
```

Имайте предвид, че механизмът за "собственост" в този случай е сравнително прост и ще се

получи грешка, ако освободите инстанция по друг начин докато тя все още присъства в списъка. Използвайте метода `Extract` за да извлечете инстанцията от него без да я освобождавате и по този начин да поемете отговорността за освобождаването й.

В **Castle Game Engine**: Наследниците на `TX3DNode` имат автоматично управление на заетата памет когато са вмъкнати като children на друг `TX3DNode`. Основният X3D възел, `TX3DRootNode`, на свой ред обикновено се притежава от `TCastleSceneCore`. Някои други обекти също имат прост механизъм за собственост - потърсете параметри и свойства, наречени `OwnsXxx`.

5.4. Виртуалният деструктор Destroy

Както видяхте в примерите по-горе, когато класът се унищожожава, се извиква неговият `деструктор`, наречен `Destroy`.

На теория можете да имате много деструктори, но на практика почти никога не е добра идея. Много по-лесно е да имате само един деструктор, наречен `Destroy`, който от своя страна се извиква от метода `Free`, той пък от своя страна се извиква от процедурата `FreeAndNil`.

Деструкторът `Destroy` в `TObject` е дефиниран като *виртуален* метод, така че винаги трябва да го маркирате с ключовата дума `override` във всички ваши класове (тъй като всички класове произлизат от `TObject`). Това е предпоставка за правилната работа на `Free`. Спомнете си как работят виртуалните методи от Виртуални методи, подмяна и скриване.

Тази информация за *деструкторите* не важи за *конструкторите*.

Нормално е един клас да има множество конструктори. Обикновено всички те се наричат `Create` и имат различни параметри, но не е неправилно да има и конструктори с други имена.

Освен това конструкторът `Create` в `TObject` *не е виртуален*, така че не го маркирайте с `override` в наследниците.

Това дава допълнителна гъвкавост при дефиниране на конструкторите. Често не е необходимо те да са виртуални, така че по подразбиране не сте принудени да го правите.

ЗАБЕЛЕЖКА

Имайте предвид обаче, че ситуацията е различна за наследниците на `TComponent`. `TComponent` дефинира виртуален конструктор `Create(AOwner: TComponent)` защото се нуждае от такъв, за да работи системата за сериализация. Когато наследявате `TComponent`, трябва да замените този конструктор (да го маркирате с ключовата дума `override`) и да извършите цялата си инициализация вътре в него. Дефинирането на допълнителни конструктори все пак е възможно, но те трябва да са само в ролята на *"помощни"*. Инстанцията трябва да работи винаги, когато е създадена с помощта *именно* на конструктора `Create(AOwner: TComponent)`, в противен случай тя няма да бъде правилно конструирана при сериализацията. *Сериализация* се използва например когато записвате и зареждате този компонент във форма на Lazarus.

5.5. Известие при освобождаване

Ако копирате препратка към инстанция, така че да имате две препратки към една и съща памет, и след това едната от тях се освободи — другата се превръща във *"висящ указател"*. Тя не бива да се използва, тъй като сочи към памет, която вече не е заета. Достъпът до нея може да доведе до грешка по време на изпълнение или връщане на произволен "боклук" (тъй като паметта може да се използва повторно вече за други неща).

Използването на `FreeAndNil` тук вече не може да помогне. `FreeAndNil` записва `nil` само в препратката, която е получила — няма начин да нулира автоматично всички други препратки. Разгледайте следния код:

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

  // какво ще се случи ако достъпим тук Obj1 или Obj2?
end;
```

1. В края на този блок `Obj1` е `nil`. Ако някакъв код трябва да има достъп до него, той може надеждно да използва `if Obj1 <> nil then ⋯`, за да избегне извикване на методи на несъществуваща вече инстанция, като

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

   Опитът за достъп до поле на нулева инстанция води до предвидимо изключение по време на изпълнение. Така че дори ако някой код не провери `Obj1 <> nil` и сляпо се обърне към полето `Obj1`, ще се получи ясно изключение по време на изпълнение.

   Същото важи и за извикване на виртуален или невиртуален метод, който се обръща към поле на `nil` инстанция.

2. С `Obj2`, нещата не са така предвидими. Той не е `nil`, но е невалиден. Опита за обръщение към поле на ненулева невалидна инстанция ще предизвика непредвидимо поведение — може би ще предизвика изключение (exception), а може би ще върне безсмислени данни.

Има различни решения на проблема:

- Едно от решенията е да бъдете внимателни и да прочетете документацията. Не предполагайте нищо относно живота на инстанцията, ако е създадена от друг код. Ако клас `TCar` има поле, сочещо към някакъв екземпляр на `TWheel`, *конвенцията* е, че препратката към *wheel* е валидна, докато препратката към *car* съществува, и *car* ще

освободи своите *wheels* в своя деструктор. Но това е само конвенция, документацията трябва да споменава, ако има нещо по-сложно.

- В горния пример, веднага след освобождаването на екземпляра `Obj1`, можете просто да присвоите изрично `nil` на променливата `Obj2` . Това е тривиално в такива прости случаи.
- Най-сигурното решение е да се използва механизма на клас `TComponent` за "известяване при освобождаване". Един компонент може да бъде известен, когато друг току-що е освободен и по този начин неговата референция да се направи равна на `nil`.

  По този начин получавате нещо подобно на *слаба референция*. Тя може да се справи в различни сценарии, например можете да оставите кода извън класа да зададе вашата препратка, а външният код може също да освободи екземпляра по всяко време.

  Това изисква и двата класа да са наследници на `TComponent`. Използването му като цяло се свежда до извикване на `FreeNotification`, `RemoveFreeNotification` и замяна на `Notification`.

  Ето пълен пример, показващ как да използвате този механизъм, заедно с конструктор/деструктор и свойство за настройка със *setter*. Понякога може да се направи и по-просто, но това е пълната версия, която винаги е правилна :)

```
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
    { set to nil by SetSomeSpecialControl to clean nicely }
    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
  { set to nil by SetSomeSpecialControl, to detach free notification }
  SomeSpecialControl := nil;
  inherited;
end;
```

5.6. Наблюдател за известие при освобождаване (Castle Game Engine)

В **Castle Game Engine** препоръчваме да използвате `TFreeNotificationObserver` от модула `CastleClassUtils` вместо директно извикване на `FreeNotification`, `RemoveFreeNotification` и замяна на `Notification`.

Като цяло използването на `TFreeNotificationObserver` изглежда малко по-просто от използването на механизма `FreeNotification` директно (въпреки че признавам, че е въпрос на вкус). Но по-специално, когато *един и същи екземпляр на клас трябва да се наблюдава поради множество причини* тогава `TFreeNotificationObserver` е много по-прост за използване (директното използване на `FreeNotification` в този случай може да стане комплицирано, тъй като трябва да внимавате да не дерегистрирате известието твърде скоро) .

Това е примерният код, използващ `TFreeNotificationObserver`, за постигане на същия ефект като примера в предишния раздел:

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

  TContainer = class(TComponent)
  private
    FSomeSpecialControlObserver: TFreeNotificationObserver;
    FSomeSpecialControl: TControl;
    procedure SetSomeSpecialControl(const Value: TControl);
    procedure SomeSpecialControlFreeNotification(const Sender:
TFreeNotificationObserver);
  public
    constructor Create(AOwner: TComponent); override;
    property SomeSpecialControl: TControl
      read FSomeSpecialControl write SetSomeSpecialControl;
```

```
  end;

implementation

uses CastleComponentSerialize;

constructor TContainer.Create(AOwner: TComponent);
begin
  inherited;
  FSomeSpecialControlObserver := TFreeNotificationObserver.Create(Self);
  FSomeSpecialControlObserver.OnFreeNotification := {$ifdef FPC}@{$endif}
SomeSpecialControlFreeNotification;
end;

procedure TContainer.SetSomeSpecialControl(const Value: TControl);
begin
  if FSomeSpecialControl <> Value then
  begin
    FSomeSpecialControl := Value;
    FSomeSpecialControlObserver.Observed := Value;
  end;
end;

procedure TContainer.SomeSpecialControlFreeNotification(const Sender:
TFreeNotificationObserver);
begin
  // set property to nil when the referenced component is freed
  SomeSpecialControl := nil;
end;
```

Вижте https://castle-engine.io/custom_components .

6. Изключения

6.1. Преглед

Изключенията позволяват *прекъсване на нормалния ход на изпълнение на кода*.

- Във всеки момент от програмата можете да **предизвикате** изключение, като използвате ключовата дума `raise`. На практика редовете код, следващи извикването `raise ⋯`, няма да се изпълнят.
- Изключение може да бъде **прихванато** с помощта на конструкция `try ⋯ except ⋯ end`. Прихващането на изключение означава, че по някакъв начин ще се "справите" с изключението и следващият код трябва да се изпълни както обикновено, изключението повече няма да се разпространява нагоре.

  Забележка: Ако бъде предизвикано изключение, но то никога не е уловено, това ще доведе до спиране на цялото приложение с грешка.

    - Но в LCL приложенията изключенията около събития (events) винаги се улавят (и извеждат в LCL диалогов прозорец), ако предварително не ги прихванете.
    - В *Castle Game Engine* приложения, използващи `CastleWindow`, изключенията около вашите събития винаги се прихващат по същия начин (и се показва правилния диалогов прозорец).
    - Така че не е толкова лесно да се предизвика изключение, което *не е прихванато никъде* (не е прихванато във вашия код, в LCL код, в CGE код...).
- Въпреки че изключенията прекъсват изпълнението, можете да използвате конструкцията `try ⋯ finally ⋯ end`, за да изпълните някакъв код **винаги**, дори ако кодът е бил прекъснат от изключение.

  Конструкцията `try ⋯ finally ⋯ end` също сработва, когато кодът е прекъснат от ключови думи `Break` или `Continue` или `Exit`. Въпросът е кода в секцията `finally` да се изпълнява наистина *винаги*.

Като цяло "изключение" може да бъде инстанция от всеки един клас.

- Компилаторът не налага никой конкретен клас. Просто трябва да извикате `raise XXX`, където `XXX` е екземпляр от какъвто и да е клас (така че всичко, произлизащо от `TObject` става за целта).
- Стандартна конвенция за класовете от изключения е те да наследяват специалния клас `Exception`. Класът `Exception` наследява `TObject`, като добавя свойството низ `Message` и конструктор за лесно задаване на това свойство. Всички изключения, предизвикани от стандартната библиотека, наследяват `Exception`. Съветваме ви да следвате тази конвенция.
- Класовете с изключение (по конвенция) имат имена, които започват с `E`, не с `T`. Например `ESomethingBadHappened`.
- Компилаторът автоматично ще освободи обекта-изключение, когато той бъде обработен. Не го освобождавайте сами.

  В повечето случаи вие просто конструирате обекта по същото време, когато извиквате `raise`, например `raise ESomethingBadHappened.Create('Описание на случилото се лошо нещо.')`.

6.2. Предизвикване

Ако искате да предизвикате свое собствено изключение, декларирайте го и извикайте `raise ⋯`, когато е подходящо:

```
type
  EInvalidParameter = class(Exception);

function ReadParameter: String;
begin
  Result := Readln;
  if Pos(' ', Result) <> 0 then
    raise EInvalidParameter.Create('Invalid parameter, space is not allowed');
```

```
end;
```

Обърнете внимание, че изразът след `raise` трябва да бъде валиден екземпляр на клас. Почти винаги ще създавате екземпляра за изключение по време на предизвикването му.

Можете също да използвате конструктора `CreateFmt`, който е удобно съкращение на `Create(Format(MessageFormat, MessageArguments))`. Това е обичаен начин за предоставяне на повече информация в съобщението за изключение. Можем да подобрим предишния пример така:

```
type
  EInvalidParameter = class(Exception);

function ReadParameter: String;
begin
  Result := Readln;
  if Pos(' ', Result) <> 0 then
    raise EInvalidParameter.CreateFmt('Невалиден параметър %s, не са позволени
интервали.', [Result]);
end;
```

6.3. Прихващане

Можете да прихванете изключение така:

```
var
  Parameter1, Parameter2, Parameter3: String;
begin
  try
    Writeln('Въведете 1-ви параметър:');
    Parameter1 := ReadParameter;
    Writeln('Въведете 2-ри параметър:');
    Parameter2 := ReadParameter;
    Writeln('Въведете 3-ти параметър:');
    Parameter3 := ReadParameter;
  except
    // прихващане на EInvalidParameter предизвикан от някое от извикванията на
ReadParameter
    on EInvalidParameter do
      Writeln('Възникна изключение EInvalidParameter');
  end;
end;
```

За да подобрим горния пример, можем да декларираме име за инстанцията на изключение (ще използваме `E` в примера). По този начин можем да отпечатаме съобщението за грешка:

```
try
...
except
  on E: EInvalidParameter do
    Writeln('Възникна изключение EInvalidParameter със съобщение: ' + E.Message);
end;
```

Може също да се тества за множество изключения:

```
try
...
except
  on E: EInvalidParameter do
    Writeln('Възникна изключение EInvalidParameter със съобщение: ' + E.Message);
  on E: ESomeOtherException do
    Writeln('Възникна изключение ESomeOtherException със съобщение: ' + E.Message);
end;
```

Можете също така да отработите и произволно предизвикано изключение, ако не използвате никакъв израз `on`:

```
try
...
except
  Writeln('Предупреждение: Възникна изключение');
end;
// ПРЕДУПРЕЖДЕНИЕ: НЕ СЛЕДВАЙТЕ ПРИМЕРА БЕЗ ДА СТЕ ПРОЧЕЛИ ЗАБЕЛЕЖКАТА ПО-ДОЛУ
// ОТНОСНО "ПРИХВАЩАНЕ НА ВСИЧКИ ИЗКЛЮЧЕНИЯ"
```

Като цяло *трябва да прихванете само изключения от определен клас, които сигнализират за определен проблем, с който знаете как да се справите*. Бъдете внимателни с прихващането на изключения от общ тип (като всяко `Exception` или всеки `TObject`), тъй като лесно можете да уловите прекалено много и по-късно да причините проблеми при отстраняване на други грешки. Както във всички езици за програмиране с изключения, доброто правило, което трябва да следвате, е *никога да не прихващате изключение, с което не знаете как да се справите*. По-специално, не прихващайте изключение само за да отстраните проблема, без първо да проучите *защо* възниква изключението.

- Изключението показва ли проблем при въвеждането от потребителя? Тогава трябва да го докладвате на потребителя.
- Изключението показва ли грешка във вашия код? Тогава трябва да поправите кода, за да не се случва повече изключението.

Друг начин да прихванете всички изключения е да използвате:

```
try
```

```
...
except
  on E: TObject do
    Writeln('Предупреждение: Възникна изключение');
end;
// ПРЕДУПРЕЖДЕНИЕ: НЕ СЛЕДВАЙТЕ ПРИМЕРА БЕЗ ДА СТЕ ПРОЧЕЛИ ЗАБЕЛЕЖКАТА ПО-ГОРЕ
// ОТНОСНО "ПРИХВАЩАНЕ НА ВСИЧКИ ИЗКЛЮЧЕНИЯ"
```

Въпреки че обикновено е достатъчно да се прихване само `Exception`:

```
try
...
except
  on E: Exception do
    Writeln('Предупреждение: Възникна изключение: ' + E.ClassName + ', съобщение: ' +
E.Message);
end;
// ПРЕДУПРЕЖДЕНИЕ: НЕ СЛЕДВАЙТЕ ПРИМЕРА БЕЗ ДА СТЕ ПРОЧЕЛИ ЗАБЕЛЕЖКАТА ПО-ГОРЕ
// ОТНОСНО "ПРИХВАЩАНЕ НА ВСИЧКИ ИЗКЛЮЧЕНИЯ"
```

Можете да "предизвикате отново" изключението в блока `except ⋯ end`, ако е необходимо. Можете да извикате `raise E;`, ако инстанцията е `E`, можете също така просто да използвате `raise` без параметър. Например:

```
try
...
except
  on E: EInvalidSoundFile do
  begin
    if E.InvalidUrl = 'http://example.com/blablah.wav' then
      Writeln('Предупреждение: зареждането на http://example.com/blablah.wav се
провали, игнорирайте го')
    else
      raise;
  end;
end;
```

Имайте предвид, че въпреки че изключението е екземпляр на обект, никога не бива да го освобождавате ръчно. Компилаторът ще генерира подходящ код, който гарантира освобождаването след като бъде обработен.

6.4. Finally (изпълнение на код независимо дали има изключение)

Често се използва конструкцията `try .. finally .. end`, за освобождаване на екземпляр от някакъв клас, независимо дали е възникнало изключение при използването му. Начинът

за използване е следния:

```
procedure MyProcedure;
var
  MyInstance: TMyClass;
begin
  MyInstance := TMyClass.Create;
  try
    MyInstance.DoSomething;
    MyInstance.DoSomethingElse;
  finally
    FreeAndNil(MyInstance);
  end;
end;
```

Това работи надеждно винаги и не причинява изтичане на памет, дори ако `MyInstance.DoSomething` или `MyInstance.DoSomethingElse` предизвикат изключение.

Обърнете внимание, че това взема предвид, че локалните променливи, като `MyInstance` по-горе, имат недефинирани стойности (може да съдържат случаен "боклук в паметта") преди първото присвояване. Тоест, писането на нещо подобно *не* би било вярно:

```
// НЕКОРЕКТЕН ПРИМЕР:
procedure MyProcedure;
var
  MyInstance: TMyClass;
begin
  try
    CallSomeOtherProcedure;
    MyInstance := TMyClass.Create;
    MyInstance.DoSomething;
    MyInstance.DoSomethingElse;
  finally
    FreeAndNil(MyInstance);
  end;
end;
```

Горният пример е грешен: ако възникне изключение в `TMyClass.Create` (конструктора може също да предизвика изключение) или в рамките на `CallSomeOtherProcedure`, тогава променливата `MyInstance` не се инициализира. Извикването на `FreeAndNil(MyInstance)` ще се опита да извика деструктора на `MyInstance`, което най-вероятно ще се срине с *Access Violation (Segmentation Fault)*. Всъщност едно изключение ще причини друго изключение, което ще направи съобщението за грешка безполезно: няма да видите съобщението на първото изключение.

Понякога е оправдано да поправите горния код, като първо инициализирате всички локални променливи на `nil` (тогава извикването на `FreeAndNil` е безопасно). Това има смисъл, ако освобождавате **много** екземпляри на класове. Така че двата примера по-долу

работят еднакво добре:

```
procedure MyProcedure;
var
  MyInstance1: TMyClass1;
  MyInstance2: TMyClass2;
  MyInstance3: TMyClass3;
begin
  MyInstance1 := TMyClass1.Create;
  try
    MyInstance1.DoSomething;

    MyInstance2 := TMyClass2.Create;
    try
      MyInstance2.DoSomethingElse;

      MyInstance3 := TMyClass3.Create;
      try
        MyInstance3.DoYetAnotherThing;
      finally
        FreeAndNil(MyInstance3);
      end;
    finally
      FreeAndNil(MyInstance2);
    end;
  finally
    FreeAndNil(MyInstance1);
  end;
end;
```

Вероятно това е по-четливо във вида по-долу:

```
procedure MyProcedure;
var
  MyInstance1: TMyClass1;
  MyInstance2: TMyClass2;
  MyInstance3: TMyClass3;
begin
  MyInstance1 := nil;
  MyInstance2 := nil;
  MyInstance3 := nil;
  try
    MyInstance1 := TMyClass1.Create;
    MyInstance1.DoSomething;

    MyInstance2 := TMyClass2.Create;
    MyInstance2.DoSomethingElse;

    MyInstance3 := TMyClass3.Create;
```

```
    MyInstance3.DoYetAnotherThing;
  finally
    FreeAndNil(MyInstance3);
    FreeAndNil(MyInstance2);
    FreeAndNil(MyInstance1);
  end;
end;
```

ЗАБЕЛЕЖКА	В този прост пример можете да направите правилния довод, че кодът би трябвало да се раздели на 3 отделни процедури, като едната извиква всяка от другите две.

6.5. Как изключенията се показват от различните библиотеки

- В случая на Lazarus LCL, изключенията, предизвикани по време на събития (различни обратни извиквания, *callbacks*, присвоени на свойствата на `OnXxx` в LCL компонентите) ще бъдат прихванати и ще доведат до диалогово съобщение, което позволява на потребителя да продължи или да спре приложението. Това означава, че вашите собствени изключения не "излизат" от `Application.ProcessMessages`, така че те не прекъсват директно приложението. Можете да конфигурирате какво точно да се случи с помощта на `TApplicationProperties.OnException`.
- По същия начин, в *Castle Game Engine* с `CastleWindow`: изключението се прихваща вътрешно и води до съобщение за грешка. Така изключенията не "излизат" от `Application.ProcessMessages`. Отново можете да конфигурирате какво да се случва с помощта на `Application.OnException`.
- Други GUI библиотеки може да направят нещо подобно на горното.
- В случай на други приложения, можете да конфигурирате как се показва изключението, като присвоите глобален *callback* на `OnHaltProgram`.

7. Run-time библиотека

7.1. Вход/изход с помощта на потоци

Съвременните програми на Паскал трябва да използват класа `TStream` и неговите наследници за да извършват входно/изходни операции. Много полезни класове наследяват `TStream`, например: `TFileStream`, `TMemoryStream`, `TStringStream`.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

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('Read from file got integer: ', OutputInt);
end.
```

В **Castle Game Engine**: Трябва да използвате функцията `Download` за създаването на поток, който получава данни от произволен URL адрес. По този начин се поддържат обикновени файлове, HTTP и HTTPS ресурси, Android assets и други. Освен това, за да отворите ресурс във вашите данни за играта (в поддиректорията `data`), използвайте специалния URL адрес `castle-data:/xxx`. Примери:

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

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

```
S := Download('castle-data:/gui/my_image.png');
```

За да четете текстови файлове, препоръчваме да използвате класа `TCastleTextReader`. Той предоставя поредово API и съдържа в себе си `TStream`. Конструкторът `TCastleTextReader` може да вземе готов URL адрес или вие можете да подадете там вашия персонализиран източник `TStream`.

```
Text := TCastleTextReader.Create('castle-data:/my_data.txt');
try
  while not Text.Eof do
    WriteLnLog('NextLine', Text.ReadLn);
finally
```

```
  FreeAndNil(Text);
end;
```

7.2. Контейнери (списъци, речници), използващи генерици

Езикът и run-time библиотеката предлагат различни гъвкави контейнери. Има редица "негенерични" класове (като `TList` и `TObjectList` от модула `Contnrs`), има и динамични масиви (`array of TMyType`). Но за да получите най-голяма гъвкавост **и** безопасност, съветвам за повечето от вашите нужди да използвате **генерични контейнери**.

Генеричните контейнери ви дават много полезни методи за добавяне, премахване, обхождане, търсене, сортиране... Компилаторът също така знае (и проверява), че контейнерът съдържа единствено елементи от указания тип.

В момента има три библиотеки, предоставящи генерични контейнери в FPC:

- Модул `Generics.Collections` (от FPC >= 3.2.0)
- Модул `FGL`
- Модул `GVector` (включен в `fcl-stl`)

Съветваме да се използва модул `Generics.Collections`. Генеричните контейнери реализирани там са:

- пакетирани с полезни функции,
- много ефективни (особено важно при достъп до речници[5] с помощта на ключове),
- съвместими между FPC и Delphi,
- именуването е в съответствие с другите части на стандартната библиотека (като негенеричните контейнери от модула `Contnrs`).

В **Castle Game Engine**: Ние използваме интензивно `Generics.Collections` и съветваме да използвате `Generics.Collections` и във вашите приложения!

Най-важните класове от `Generics.Collections` са:

TList

Генеричен списък от елементи от указан тип.

TObjectList

Генеричен списък от екземпляри от указан клас. Може да "притежава" екземплярите, което означава че ще ги унищожи автоматично при унищожаване на списъка.

TDictionary

Генеричен речник[5].

TObjectDictionary

Генеричен речник, Може да "притежава" ключовете и/или стойностите.

Ето как да използвате прост генеричен `TObjectList`:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses SysUtils, Generics.Collections;

type
  TApple = class
    Name: string;
  end;

  TAppleList = {$ifdef FPC}specialize{$endif} TObjectList<TApple>;

var
  A: TApple;
  Apples: TAppleList;
begin
  Apples := TAppleList.Create(true);
  try
    A := TApple.Create;
    A.Name := 'my apple';
    Apples.Add(A);

    A := TApple.Create;
    A.Name := 'another apple';
    Apples.Add(A);

    Writeln('Count: ', Apples.Count);
    Writeln(Apples[0].Name);
    Writeln(Apples[1].Name);
  finally FreeAndNil(Apples) end;
end.
```

Обърнете внимание, че някои операции изискват сравняване на два елемента, като сортиране и търсене (напр. чрез методите `Sort` и `IndexOf`). Контейнерите в `Generics.Collections` използват за това *сравнител*. *Подразбиращия се сравнител* е смислен за всички типове, дори за записи (в дадения случай сравнява съдържанието на паметта, което е разумна настройка по подразбиране поне за търсене чрез `IndexOf`).

Когато сортирате списък, можете да укажете *персонализиран сравнител* като параметър. *Сравнителя* е клас, реализиращ интерфейса `IComparer`. На практика обикновено дефинирате подходящ *callback* и използвате метода `TComparer<T>.Construct`, за да пакетирате този *callback* в екземпляр на `IComparer`. Пример за това е по-долу:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
```

```
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

{ If GENERICS_CONSTREF is defined, then various routines used with
Generics.Collections
  (like callbacks we pass to TComparer, or OnNotify callback or Notify virtual method)
  should have "constref" parameter, not "const".
  This was the case in FPC <= 3.2.0.
  FPC changed constref->const in
  https://gitlab.com/freepascal.org/fpc/source/-
/commit/693491048bf2c6f9122a0d8b044ad0e55382354d
  and the change is present in FPC fixes branch 3.2.3 and later 3.2.4(rc1). }
{$ifdef VER3_0} {$define GENERICS_CONSTREF} {$endif}
{$ifdef VER3_2_0} {$define GENERICS_CONSTREF} {$endif}
{$ifdef VER3_2_2} {$define GENERICS_CONSTREF} {$endif}

uses SysUtils, Generics.Defaults, Generics.Collections;

type
  TApple = class
    Name: string;
  end;

  TAppleList = {$ifdef FPC}specialize{$endif} TObjectList<TApple>;

function CompareApples(
  {$ifdef GENERICS_CONSTREF}constref{$else}const{$endif}
  Left, Right: TApple): Integer;
begin
  Result := AnsiCompareStr(Left.Name, Right.Name);
end;

type
  TAppleComparer = {$ifdef FPC}specialize{$endif} TComparer<TApple>;
var
  A: TApple;
  L: TAppleList;
begin
  L := TAppleList.Create(true);
  try
    A := TApple.Create;
    A.Name := '11';
    L.Add(A);

    A := TApple.Create;
    A.Name := '33';
    L.Add(A);

    A := TApple.Create;
    A.Name := '22';
    L.Add(A);
```

```
    L.Sort(TAppleComparer.Construct({$ifdef FPC}@{$endif} CompareApples));

    Writeln('Count: ', L.Count);
    Writeln(L[0].Name);
    Writeln(L[1].Name);
    Writeln(L[2].Name);
  finally FreeAndNil(L) end;
end.
```

Класът `TDictionary` реализира **речник**, познат като **map (key → value)**, също познат като **associative array**. Неговото API е подобно на `TDictionary` в C#. Има полезни итератори за ключове, стойности и двойки ключ→стойност.

Примерен код, използващ речник:

```
{$mode objfpc}{$H+}{$J-}
uses SysUtils, Generics.Collections;

type
  TApple = class
    Name: string;
  end;

  TAppleDictionary = {$ifdef FPC}specialize{$endif} TDictionary<String, TApple>;

var
  Apples: TAppleDictionary;
  A, FoundA: TApple;
  ApplePair: {$ifdef FPC} TAppleDictionary.TDictionaryPair {$else} TPair<String,
TApple> {$endif};
  AppleKey: string;
begin
  Apples := TAppleDictionary.Create;
  try
    A := TApple.Create;
    A.Name := 'моята ябълка';
    Apples.AddOrSetValue('ключ за ябълка 1', A);

    if Apples.TryGetValue('ключ за ябълка 1', FoundA) then
      Writeln('Намерена ябълка с ключ "ключ за ябълка 1" с име: ' +
        FoundA.Name);

    for AppleKey in Apples.Keys do
      Writeln('Намерен ключ за ябълка: ' + AppleKey);
    for A in Apples.Values do
      Writeln('Намерена ябълка с име: ' + A.Name);
    for ApplePair in Apples do
      Writeln('Намерен ключ за ябълка->име на ябълка: ' +
        ApplePair.Key + '->' + ApplePair.Value.Name);
```

```
    { Долният ред също работи, но може да се използва само да
      зададе стойност на *съществуващ* ключ в речника.
      Вместо това обикновено се използва AddOrSetValue
      за да се зададе или добави нов ключ ако е необходимо. }
    // Apples['ключ за ябълка 1'] := ... ;

    Apples.Remove('ключ за ябълка 1');

    { Забележете, че TDictionary не притежава елементите си
      и трябва да ги освобождавате ръчно.
      Може да използвате TObjectDictionary за да имате автоматичен
      режим за притежание. }
    A.Free;
  finally FreeAndNil(Apples) end;
end.
```

`TObjectDictionary` може да *притежава* ключовете и/или стойностите, което означава че ще ги унищожава автоматично. Внимавайте *това притежание да бъде само когато ключовете/стойностите са екземпляри на обекти*. Ако укажете, че ще се притежават елементи от друг тип, например `Integer` (т.е. ако ключовете са `Integer`, и включите `doOwnsKeys`), ще получите много неприятен срив при изпълнение на програмата.

Пример за това как се използва `TObjectDictionary` е даден по-долу. Компилирайте примера с *memory leak detection*, напр. така `fpc -gl -gh generics_object_dictionary.dpr`, за да видите, че няма изтичане на памет при приключване на програмата.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

uses SysUtils, Generics.Collections;

type
  TApple = class
    Name: string;
  end;

  TAppleDictionary = {$ifdef FPC}specialize{$endif} TObjectDictionary<String, TApple>;

var
  Apples: TAppleDictionary;
  A: TApple;
  ApplePair: {$ifdef FPC} TAppleDictionary.TDictionaryPair {$else} TPair<String,
TApple> {$endif};
begin
  Apples := TAppleDictionary.Create([doOwnsValues]);
  try
    A := TApple.Create;
    A.Name := 'my apple';
    Apples.AddOrSetValue('apple key 1', A);
```

```
    for ApplePair in Apples do
      Writeln('Found apple key->value: ' +
        ApplePair.Key + '->' + ApplePair.Value.Name);

    Apples.Remove('apple key 1');
  finally FreeAndNil(Apples) end;
end.
```

Ако предпочитате да използвате модула `FGL` вместо `Generics.Collections`, най-важните класове от `FGL` са:

TFPGList

Генеричен списък от елементи от указан тип.

TFPGObjectList

Генеричен списък от екземпляри от указан клас. Може да "притежава" екземплярите.

TFPGMap

Генеричен речник[5].

В модул `FGL`, `TFPGList` може да се използва само с типове, които имат дефиниран оператор за равенство (=). При `TFPGMap` за типа на ключа трябват дефинирани оператори *"по-голямо"* (>) и *"по-малко"* (<). Ако искате да използвате тези контейнери с типове, които нямат дефинирани оператори за сравнение (например записи), ще трябва да им дефинирате съответните оператори както е показано в Замяна на оператори.

Ако искате да научите повече за генериците, вижте Генерици.

7.3. Клониране: TPersistent.Assign

Копирането на екземплярите на клас чрез прост оператор за присвояване `:=` копира единствено **препратката**.

```
var
  X, Y: TMyObject;
begin
  X := TMyObject.Create;
  Y := X;
  // X и Y сега са два указателя към една и съща инстанция
  Y.MyField := 123; // ще се промени също и X.MyField
  FreeAndNil(X);
end;
```

За да копирате **съдържанието на екземпляр** от някакъв клас, стандартния подход е да наследите класа от `TPersistent`, и да подмените неговия метод `Assign`. След като той бъде коректно написан за `TMyObject`, той може да се използва по следния начин:

```
var
  X, Y: TMyObject;
begin
  X := TMyObject.Create;
  Y := TMyObject.Create;
  Y.Assign(X);
  Y.MyField := 123; // това не променя X.MyField
  FreeAndNil(X);
  FreeAndNil(Y);
end;
```

За да работи правилно, кодът в тялото на метода `Assign` трябва да копира стойностите на необходимите полета. Трябва внимателно да кодирате `Assign`, за да копирате от класове, който може да са наследници на текущия клас.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

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; // копирайте още полета ако е необходимо ...
  end else
    { Поради това, че TMyClass е директен наследник на TPersistent,
      той извиква inherited САМО когато не знае как да обработи Source.
      Виж кометарите по-долу. }
    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; // копирайте още полета ако е необходимо
...
  end;

  { Поради това, че TMyClassDescendant има предшественик, който вече е
    заменил Assign (in TMyClass.Assign), той извиква inherited ВИНАГИ,
    за да позволи TMyClass.Assign да копира останалите полета.
    Виж кометарите по-долу за детайлно обяснение. }
  inherited Assign(Source);
end;

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

  // тест TMyClassDescendant.Assign
  CD1 := TMyClassDescendant.Create;
  CD2 := TMyClassDescendant.Create;
  try
    CD1.MyInt := 44;
    CD1.MyString := 'blah';
    CD2.Assign(CD1);
    WriteLn('CD2 state: ', CD2.MyInt, ' ', CD2.MyString);
  finally
    FreeAndNil(CD1);
    FreeAndNil(CD2);
  end;
end.
```

Понякога е по-удобно да замените метода `AssignTo` в класа източник, вместо да замените метода `Assign` в класа, на който се присвоява.

Бъдете внимателни, когато извиквате `inherited` в подменения `Assign`. Има две ситуации:

Вашият клас е пряк наследник на класа `TPersistent`. (Или не е пряк наследник на `TPersistent`, но нито един предшественик не е заменил метода `Assign`.)

В този случай вашият клас трябва да използва ключовата дума `inherited` (за извикване на `TPersistent.Assign`) *само ако не можете да се справите с присвояването във вашия код*.

Вашият клас произлиза от клас, който вече е заменил метода `Assign`.

В този случай вашият клас трябва *винаги* да използва ключовата дума `inherited` (за да извика наследения `Assign`). Като цяло извикването на `inherited` в подменени методи *обикновено* е добра идея.

За да разберете причината зад горното правило (кога трябва и кога *не трябва* да извикате `inherited` от имплементацията `Assign`) и как това е свързано с метода `AssignTo`, нека да разгледаме `TPersistent.Assign` и `TPersistent.AssignTo` реализации:

```
procedure TPersistent.Assign(Source: TPersistent);
begin
  if Source <> nil then
    Source.AssignTo(Self)
  else
    raise EConvertError...
end;

procedure TPersistent.AssignTo(Destination: TPersistent);
begin
  raise EConvertError...
end;
```

ЗАБЕЛЕЖКА

Това не е **точната** реализация в `TPersistent`. Копиран е кода на стандартната FPC библиотека, но след това е опростен, за да се скрият маловажни подробности относно съобщението за изключение.

Изводите, които можете да направите от горното са:

- *Ако нито `Assign`, нито `AssignTo` не са заменени*, извикването им ще доведе до изключение.
- Също така имайте предвид, че *няма* код в изпълнението на `TPersistent`, който автоматично да копира всички полета (или всички публикувани полета) на класовете. Ето защо трябва да направите това сами, като замените `Assign` във всички класове. Можете да използвате RTTI (информация за тип на изпълнение) за това, но за прости случаи вероятно просто ще копирате полетата ръчно.

Когато имате клас като `TApple`, вашата реализация `TApple.Assign` обикновено ще се занимава с копиране на полета, които са специфични само за класа `TApple` (не за предшественика на

`TApple`, като `TFruit`). И така, изпълнението на `TApple.Assign` обикновено проверява дали `Source is TApple` в началото, преди да копира полета, свързани с ябълка. След това извиква `inherited`, за да позволи на `TFruit` да обработва останалите полета.

Ако приемем, че сте написали `TFruit.Assign` и `TApple.Assign` по описания начин, тогава ефектът ще е следният:

- Ако подадете екземпляр `TApple` на `TApple.Assign`, той ще копира всички полета.
- Ако подадете екземпляр `TOrange` на `TApple.Assign`, той ще копира само общите полета на `TOrange` и `TApple`. С други думи - ще копира полетата дефинирани в `TFruit`.
- Ако подадете екземпляр `TWerewolf` на `TApple.Assign`, той ще предизвика изключение (защото `TApple.Assign` ще извика `TFruit.Assign`, който ще извика `TPersistent.Assign`, който ще предизвика изключение).

ЗАБЕЛЕЖКА

Запомнете, че когато наследявате `TPersistent`, по подразбиране *спецификатора за видимост* е `published`, за да се позволи сериализиране на наследниците на `TPersistent`. Не всички типове на полета и свойства са разрешени в секция `published`. Ако поради това получите грешки и не ви е грижа за сериализацията, просто променете видимостта на `public`. Вижте Нива на видимост.

8. Разни възможности на езика

8.1. Локални (вложени) подпрограми

Вътре в по-голяма *подпрограма* (функция, процедура, метод) може да се дефинира друга, помощна подпрограма.

Вложената подпрограма може свободно да достъпва (чете и записва) всички параметри подадени на външната, *както и всички нейни локални променливи*. Това е много мощно средство, което позволява да се разбие голяма подпрограма в няколко по-малки без да е необходимо голямо усилие (тъй като не е нужно да предавате цялата необходима информация в параметрите). Внимавайте да не прекалите — ако много вложени подпрограми използват (и дори променят) една и съща променлива на външната подпрограма, кодът може да стане труден за разчитане.

Долните два примера са еквивалентни:

```
function 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;
```

Друга версия, в която локалната функция `Square` осъществява директен достъп до `I`:

```
function 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;
```

Локалните процедури могат да достигнат всякаква дълбочина — което означава, че можете да дефинирате локална процедура в друга локална процедура. Така че можете да се развихрите (но моля, не ставайте *прекалено диви*, или кодът ще стане нечетлив:).

8.2. Callbacks (познати като *Събития*, също като *Указатели към функции*, също като *Процедурни променливи*)

Позволяват индиректно извикване на подпрограми чрез променлива. Променливата може да бъде присвоена по време на изпълнение, за да сочи към всяка функция *със съвпадащи типове параметри и връщани типове*.

Callback-ът може да бъде:

- Нормален, което означава, че може да сочи към всяка обикновена подпрограма (без методи и вложени подпрограми).

  ```
  {$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
  {$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

  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.
```

- Метод: декларира се с `of object` накрая.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

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({$ifdef FPC}@{$endif} C.Add);
    WriteLn('1 + 2 + 3 ... + 10 = ', C.CurrentValue);

    C.ProcessTheList({$ifdef FPC}@{$endif} C.Multiply);
    WriteLn('1 * 2 * 3 ... * 10 = ', C.CurrentValue);
  finally
    FreeAndNil(C);
  end;
end.
```

Имайте предвид, че *не можете* да предавате глобални процедури / функции като методи. Те не са съвместими. Ако ви трябва `of object` callback, но не искате да създавате екземпляр от фиктивен клас, можете да използвате Методи на класа за целта.

```
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;
```

За съжаление, ще трябва да изпишете грозното `@TMyClass(nil).Add` вместо просто `@TMyClass.Add`.

- (Евентуално) локална подпрограма: декларирайте с `is nested` в края и се уверете, че използвате директивата `{$modeswitch nestedprocvars}`. Те вървят ръка за ръка с Локални (вложени) подпрограми.

8.3. Генерици

Генериците са мощно средство във всеки съвременен език. Дефиницията на нещо (обикновено клас) може да бъде параметризирана с друг тип. Най-типичният пример е, когато трябва да създадете контейнер (списък, речник, дърво, граф...): тогава може да дефинирате *списък елементи от тип T*, и после *да го специализирате* за да получите незабавно *списък от цели числа*, *списък от низове*, *списък инстанции от клас TMyRecord* и т.н.

Генериците в Pascal работят подобно на генериците в C++. Което означава, че те се *"разширяват"* по време на специализацията, *подобно* на макроси (но са много по-безопасни от тях; например идентификаторите се откриват по време на дефиницията, а не при специализацията, така че не можете да "инжектирате" някакво неочаквано поведение при специализация). На практика това означава, че те са много бързи (могат да бъдат оптимизирани за всеки отделен тип) и работят с типове от всякакъв размер. Когато специализирате генеричен тип можете да използвате примитивен тип (цяло число, float), както запис, така и клас.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

{$ifndef FPC}
  {$message warn 'Delphi does not allow addition on types that are generic
parameters'}
  begin end.
{$endif}

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 = {$ifdef FPC}specialize{$endif} TMyCalculator<Single>;
```

```
  TMyStringCalculator = {$ifdef FPC}specialize{$endif} TMyCalculator<string>;

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

  StringCalc := TMyStringCalculator.Create;
  try
    StringCalc.Add('something');
    StringCalc.Add(' more');
    WriteLn('StringCalc: ', StringCalc.Value);
  finally
    FreeAndNil(StringCalc);
  end;
end.
```

Генериците не се ограничават до класове, можете да имате също генерични функции и процедури:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

{$ifndef FPC}
  {$message warn 'Delphi does not support global generic functions'}
  begin end.
{$endif}

uses SysUtils;

{ Note: this example requires FPC 3.1.1 (will not compile with FPC 3.0.0 or older). }

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.
```

Вижте също Контейнери (списъци, речници), използващи генерици относно важните стандартни класове, използващи генерици.

8.4. Overloading

Позволени са методи (също и глобални функции и процедури) с едно и също име, стига да имат различни параметри. По време на компилиране компилаторът открива кой вариант искате да използвате, като узнае параметрите, които подавате.

По подразбиране overloading-ът използва FPC подхода, което означава, че всички методи в дадено пространство от имена (клас или unit) са равнопоставени и закриват другите методи в пространства от имена с по-малък приоритет. Например, ако дефинирате клас с методи `Foo(Integer)` и `Foo(string)` и той наследява клас с метод `Foo(Float)`, тогава потребителите на вашия нов клас няма да имат достъп до метод `Foo(Float)` толкова лесно (те все още могат --- ако преобразуват класа към неговия тип-предшественик). За да преодолеете това, използвайте ключовата дума `overload`.

8.5. Препроцесор

Можете да използвате прости препроцесорни директиви за:

- условна компилация (код зависим от платформата или други ръчно зададени параметри),
- да включите един файл в друг,
- да дефинирате макроси без параметри.

Имайте предвид, че макроси с параметри не се поддържат. Като цяло трябва да избягвате използването на препроцесорните директиви... освен ако наистина не се налага. Предварителната обработка се прави преди компилатора да извърши анализа на кода, което означава, че можете да "нарушите" нормалния синтаксис на езика Pascal. Това е мощна, но и донякъде "нечиста" функция.

```
unit PreprocessorStuff;

{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}

interface

{$ifdef FPC}
{ Това е дефинирано само ако се компилира с FPC, не с други компилатори (напр.
Delphi). }
procedure Foo;
{$endif}
```

```
{ Дефиниране на константата NewLine. Тук може да видите как нормалния синтаксис на
Паскал
  се "чупи" с препроцесорните директиви. Когато компилирате за Unix
  (вкл. Linux, Android, Mac OS X), компилатора вижда това:

    const NewLine = #10;

  Когато компилирате за Windows, компилатора вижда това:

    const NewLine = #13#10;

  За други операционни системи, кодът няма да се компилира,
  защото компилатора вижда това:

    const NewLine = ;

  *Хубаво е*, че компилирането се проваля в този случай -- така ако трябва да
  пригодите програмата към ОС, която не е Unix или Windows, компилатора ще ви
  припомни да изберете конвенция за нов ред (newline) за тази система. }

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 е съкращение за $include
{$I some_other_file.inc}

end.
```

Включваните файлове обикновено имат разширение `.inc` и се използват за две цели:

- Включеният файл може да съдържа само други директиви на компилатора, които "конфигурират" вашия изходен код. Например можете да създадете файл `myconfig.inc` със следното съдържание:

```
{$ifdef FPC}
  {$mode objfpc}
  {$H+}
  {$J-}
  {$modeswitch advancedrecords}
  {$ifdef VER2}
    {$message fatal 'This code can only be compiled using FPC version >= 3.0.'}
  {$endif}
{$endif}
```

  Сега можете да включите този файл с помощта на `{$I myconfig.inc}` във всички ваши изходни файлове.

- Друга цел е да се раздели голям unit на много файлове, като същевременно се запази като един unit относно езиковите правила. Не прекалявайте с тази техника - първият ви инстинкт трябва да бъде да разделите един unit на множество unit-и, а не да разделяте един unit на множество включени файлове. Все пак това е полезна техника. Позволява да се избегне "експлозията" на броя на unit-ите, като същевременно поддържа вашите файлове с изходен код кратки. Например, може да е по-добре да имате единичен unit с *"често използвани UI контроли"* отколкото да създавате *по един unit за всеки UI контролен клас*, тъй като последното би направило клаузата `uses` дълга (тъй като обикновено UI ще зависи от няколко UI класа). Но поставянето на всички тези UI класове в един файл `myunit.pas` би го направило също така дълъг и неудобен за навигация, така че разделянето му на множество включени файлове може да има смисъл.

  1. Позволява лесно да имате междуплатформен интерфейс на unit с платформено-зависима реализация. По принцип можете да направите:

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

     Понякога това е по-добре от писането на дълъг код с много `{$ifdef UNIX}`, `{$ifdef MSWINDOWS}`, примесени с нормален код (декларации на променливи, тела на подпрограми). По този начин кодът става по-четлив. Можете дори да използвате тази техника по-агресивно, като използвате опцията на командния ред `-Fi` на FPC, за да включите някои поддиректории само за определени платформи. Тогава можете да имате много версии на включения файл `{$I my_platform_specific_implementation.inc}` и просто да ги включвате, позволявайки на компилатора да намери правилната версия.

8.6. Записи

Record е просто контейнер за други променливи. Това е като много, много опростен *class*:

няма наследяване или виртуални методи. Това е като *struct* в C-подобните езици.

Ако използвате директивата {$modeswitch advancedrecords}, записите **могат** да имат методи и спецификатори за видимост. Като цяло, тогава са възможни езикови функции, които са налични за класове и *не нарушават простото предвидимо разпределение на паметта на запис*.

```
{$ifdef FPC}
  {$mode objfpc}{$H+}{$J-}
  {$modeswitch advancedrecords}
{$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

type
  TMyRecord = record
  public
    I, Square: Integer;
    procedure WriteLnDescription;
  end;

procedure TMyRecord.WriteLnDescription;
begin
  WriteLn('Square of ', I, ' is ', 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 съвременния Обектен Паскал първият ви мисъл трябва да бъде да проектирате "клас", а не "запис" — защото класовете са пълни с полезни функции, като конструктори и наследяване.

Но записите все още са много полезни, когато имате нужда от скорост или предвидимо разпределение на паметта:

- Записите нямат конструктор или деструктор. Вие просто дефинирате променлива от тип запис. Има недефинирано съдържание (боклук от паметта) в началото (с изключение на

автоматично управлявани типове, като низове; гарантирано е, че те ще бъдат инициализирани, така че да бъдат празни, и финализирани, за да освободят броя на препратките). Така че трябва да сте по-внимателни, когато работите със записи. Те обаче ви дават известно предимство в скоростта.

- Масивите от записи са добре линеаризирани в паметта, така че са удобни за кеширане.
- Разпределението на паметта при записите (размер, празнини между полетата) е ясно дефинирано в някои ситуации: когато поискате *C layout* или когато използвате `packed record`. Това е полезно:
  - за комуникация с библиотеки, написани на други езици за програмиране, когато предоставят API, базиран на записи,
  - за четене и запис на двоични файлове,
  - да правят мръсни трикове на ниско ниво (като нерестриктирано конвертиране на типове от един тип към друг, когато сте наясно с тяхното представяне в паметта).
- Записите също могат да имат `case` варианти, които работят като *unions* в C-подобните езици. Те позволяват да се третира една и съща част от паметта като различен тип, в зависимост от вашите нужди. Това позволява по-добро използване на паметта в някои случаи. И позволява повече *мръсни, опасни трикове на ниско ниво*:)

8.7. Обекти, стар стил

Преди време Turbo Pascal въведе друг синтаксис за функционалност, подобна на клас, използвайки ключовата дума `object`. Това е донякъде смесица между концепцията за "запис" и модерната за "клас".

- Старите обекти могат да се създават / освобождават и по време на тези операции можете да извикате техния конструктор / деструктор.
- Но те също могат да и да бъдат просто декларирани и използвани, като обикновени записи. Простият тип "запис" или "обект" не е препратка (указател) към нещо друго, това са просто данни. Това ги прави удобни за малки обеми от данни, където многократното създаване и освобождаване не винаги е оправдано.
- Старите обекти предлагат наследяване и виртуални методи, макар и с малки разлики от съвременните класове. Бъдете внимателни — *лоши неща* могат да се случат, ако се опитате да използвате обект с виртуални методи, без да извикате неговия конструктор.

В повечето случаи не се препоръчва използването на обекти от стария вид. Съвременните *класове* предоставят много повече функционалност. Когато е необходимо да се повиши скоростта на изпълнение, могат да се използват записи (вкл. *разширени записи*). Този подход е по-добър от използването на стари обекти.

8.8. Указатели

Можете да създадете *указател* към всеки тип данни. Указателят към типа `TMyRecord` се декларира като `^TMyRecord` и по конвенция се нарича `PMyRecord`. По-долу е показан традиционен пример за свързан списък от цели числа, използващи записи:

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

Обърнете внимание, че дефиницията е рекурсивна (тип `PMyRecord` се дефинира с помощта на тип `TMyRecord`, докато `TMyRecord` се дефинира с помощта на `PMyRecord`). Позволено е да се дефинира тип указател към *все още недефиниран тип*, стига той да бъде дефиниран в рамките на същия раздел `type`.

Можете да заемате и освобождавате памет за указателите с помоща на методите `New` и `Dispose` или (на по-ниско ниво, типово необезопасено) методите `GetMem` и `FreeMem`. За да достъпите данните, които указателите сочат, следва да добавите оператора `^` (например `MyInteger := MyPointerToInteger^`). За да направите обратната операция, която е *получаване на указател към съществуваща променлива*, трябва да използвате префикс-оператора `@` (например `MyPointerToInteger := @MyInteger`).

Има и нетипизиран тип `Pointer`, подобен на `void*` в C-подобните езици. Той е напълно типово необезопасен и може да бъде преобразуван във всеки друг тип указател.

Не забравяйте, че *екземплярът на class* всъщност е указател, въпреки че не изисква оператори `^` или `@`, за да го използвате. Възможно е да се направи свързан списък, използващ класове, той би бил следният:

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

8.9. Замяна на оператори

Можете да замените значението на много от езиковите оператори, за да позволите например събиране и умножение във вашите потребителски типове. Като например:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

{$ifndef FPC}
  {$message warn 'Delphi does not support global operator overloading'}
  begin end.
{$endif}

uses
  StrUtils;
```

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

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

Също така можете да заменяте значението на оператори върху класове. Понеже в такива функции-оператори обикновено се създават нови екземпляри на класовете, в извикващия код трябва да се предвиди надлежното освобождаване на заетата памет.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

{$ifndef FPC}
  {$message warn 'Delphi does not support global operator overloading'}
  begin end.
{$endif}

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.
```

Можете и да замените значението на оператори върху записи. Това е по-просто отколкото да го правите върху класове, защото няма нужда да се грижите за освобождаването на заетата памет.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

{$ifndef FPC}
  {$message warn 'Delphi does not support global operator overloading'}
  begin end.
{$endif}

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.
```

За работа със записи се препоръчва да използвате `{$modeswitch advancedrecords}` и да замените операторите като `class operator` вътре в записа. Това позволява да се използват генерични контейнери, които зависят от съществуването на някакъв оператор (като `TFPGList`, който зависи от наличието на оператор за равенство) с такива записи. В противен случай "глобалната" дефиниция на оператор (която не в записа) няма да бъде открита (защото не е налична в кода, който имплементира `TFPGList`) и няма да можете да специализирате списък със `specialize TFPGList<TMyRecord>`.

```
{$ifdef FPC}
  {$mode objfpc}{$H+}{$J-}
  {$modeswitch advancedrecords}
{$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}
```

```
{$ifndef FPC}
  {$message warn 'Delphi does not have FGL unit'}
  begin end.
{$endif}

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 = {$ifdef FPC}specialize{$endif} 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.
```

9. Допълнителни възможности на класовете

9.1. Частни и лични полета

Спецификатора `private` означава, че полето (или метода) не е достъпно извън класа, в който е декларирано. Това правило обаче позволява изключение: кодът *в същия модул* може да работи с частни полета и методи. Някой програмист на C++ би могъл да каже, че *всички класове в един модул са "приятели"*[6]. Това изключение често е полезно и не нарушава енкапсулацията защото в крайна сметка е в границите на един модул.

От друга страна, ако правите големи модули с много класове, които не са силно свързани един с друг, е по-безопасно да използвате спецификатора `strict private`. Той наистина ще ограничи достъпа до полето (или метода) само в рамките на класа. Без изключения.

Аналогично — спецификатора `protected` означава, че полето или метода е достъпен за наследниците и "приятелите" в модула, докато `strict protected`, че е достъпно само за наследниците.

9.2. Допълнителни декларации и вложени класове

В един клас можете да декларирате и вложени секции за константи (`const`) или типове (`type`). По този начин може дори да се декларират и вложени класове. Спецификаторите за видимост работят както винаги, в частност вложеният клас може да бъде `private`(невидим за външния свят), което доста често е полезно.

Имайте предвид, че за да декларирате поле след константа или тип, ще трябва да започнете 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;

{ забележете, че дефиницията на метода долу има префикс
  "TMyClass.TInternalClass". }
procedure TMyClass.TInternalClass.DoSomething;
begin
end;
```

9.3. Методи на класа

Това са методи, които можете да извикате с препратка към клас (`TMyClass`), не непременно към екземпляр на клас.

```
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;
```

Имайте предвид, че те също могат да бъдат виртуални - това понякога е много полезно когато се комбинират с Препратки към клас.

Методите на клас също могат да бъдат ограничени с Нива на видимост като `private` or `protected` съвсем като обикновените методи.

Имайте предвид, че конструкторът винаги действа като метод на клас, когато се извиква по нормален начин `MyInstance := TMyClass.Create(⋯);`. Въпреки, че е възможно също така да се

извика конструктор в тялото на метод на самия клас и тогава той действа като обикновен метод. Това е полезна функция за "верижни" конструктори, когато един конструктор (напр. подменен за да приеме целочислен параметър) върши нещо и след това извиква друг конструктор (напр. без параметър).

9.4. Препратки към клас

Препратките към клас ви позволяват да изберете класа по време на изпълнение, например да извикате метод на клас или конструктор, без да знаете точния клас по време на компилация. Това е тип, деклариран като `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
  // Obviously you can do this:

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

  // В допълнение, използвайки препратки към клас, може да направите и следното:

  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;
```

Препратките към класове могат да се комбинират с виртуални клас-методи. Това дава същия ефект както използването на класове с виртуални методи - действителният метод, който трябва да бъде извикан, се определя по време на изпълнение.

```
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;

  { Това ще предизвика изключение по време на изпълнение
    защото DoSomething е абстрактен в TMyClass. }
  ClassRef := TMyClass;
  ClassRef.DoSomething;
end;
```

Ако имате екземпляр и искате да получите препратка към неговия клас (не декларирания клас, а същинския клас използван при неговото конструиране), можете да използвате свойството `ClassType`. Типа на `ClassType` е `TClass`, който е деклариран като `class of TObject`. Често можете без проблем да го преобразувате към по-конкретен клас, ако ви е известно, че е екземплярът е нещо по-специфично от `TObject`.

Можете да използвате препратката от `ClassType` за извикване на виртуални методи, в това число виртуални конструктори. Това ви позволява да създадете метод `Clone`, който създава екземпляр от *точния клас на текущия обект*. Може да го комбинирате с Клониране: TPersistent.Assign за да получите метод, който връща нов "клонинг" на инстанцията от която е извикан.

Не забравяйте, че това ще работи само когато конструкторът на вашия клас е виртуален. Например, може да се използва със стандартните наследници на `TComponent`, тъй като всички те трябва да заменят виртуалния конструктор `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
  // Това трябва винаги да създаде инстанция точно от клас TMyClass:
  //Result := TMyClass.Create(AOwner);
  // Това може потенциално да създаде инстанция от наследник на TMyClass:
  Result := TMyClassRef(ClassType).Create(AOwner);
  Result.Assign(Self);
end;
```

9.5. Статични методи на клас

За да разберете *статичните методи на клас*, трябва да разберете как работят *нормалните методи на клас* (описани в предишните раздели). Вътрешно, *нормалните методи на клас* получават *референция* към своя клас (тя се предава през скрит, неявно добавен параметър на метода). Тази препратка може да се използва с помощта на ключовата дума `Self` в метода на класа. Обикновено това е полезно: тази препратка към клас ви позволява да извиквате *виртуалните методи на класа* (чрез *таблицата с виртуални методи* на класа).

Наличието на скрита препратка обаче, прави *методите на класа* несъвместими с *процедурните променливи*. Следната програма **няма да може да се компилира**:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}

type
  TMyCallback = procedure (A: Integer);

  TMyClass = class
    class procedure Foo(A: Integer);
  end;

class procedure TMyClass.Foo(A: Integer);
begin
end;

var
  Callback: TMyCallback;
begin
  // Грешка: TMyClass.Foo не е съвместим с TMyCallback
  Callback := {$ifdef FPC} @TMyClass(nil).Foo {$else} TMyClass.Foo {$endif};
end.
```

ЗАБЕЛЕЖКА Ако сте в режим *Delphi* тогава ще можете да напишете `TMyClass.Foo`

> вместо грозното `TMyClass(nil).Foo` което е в горния пример. Трябва да се признае, че `TMyClass.Foo` изглежда много по-елегантно и също така се проверява по-добре от компилатора. Използването на `TMyClass(nil).Foo` е хак... за съжаление необходим (засега) в режима *ObjFpc*, който е представен в тази книга.
>
> Във всеки случай, присвояването на `TMyClass.Foo` на `Callback` по-горе би *било неуспешно* и в режим Delphi, поради абсолютно същите причини.

Горният пример не се компилира, защото типа на `Callback` не е съвместим с метода на класа `Foo`. Това е така, защото вътрешно методът `Foo` има този специален скрит *implicit* параметър за препратката към класа.

Един от начините да коригирате горния пример е да промените дефиницията на `TMyCallback` на следната: `TMyCallback = procedure (A: Integer) of object;`. Но понякога това не е желателно.

Другият начин е метода да се укаже като `static`. По същество такъв метод е просто глобална процедура / функция, с тази разлика, че видимостта му е ограничена вътре в класа. Той *няма* такава скрита препратка към клас (по този начин *не може да бъде виртуален* и *не може да извиква виртуални методи*). От друга страна, той е съвместим с нормалните (необектни) процедурни променливи. Така че това ще работи:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

type
  TMyCallback = procedure (A: Integer);

  TMyClass = class
    class procedure Foo(A: Integer); static;
  end;

class procedure TMyClass.Foo(A: Integer);
begin
end;

var
  Callback: TMyCallback;
begin
  Callback := @TMyClass.Foo;
end.
```

9.6. Полета и свойства на клас

Полето на клас може да се дефинира в секция `class var` вътре в класа. То е подобно на обикновеното поле но няма нужда от инстанция за да се достъпва. Като резултат, то е подобно на глобална променлива но видимостта му е ограничена само в класа, в който е

дефинирано.

Свойството на клас е такова свойство, което може да се достъпи през референция на клас и без да е необходимо да има създадена инстанция. Дефинира се с `class property` вместо само с `property` и с методи *getter* и / или *setter*, които обаче трябва да са *статични клас-методи*. Виж Статични методи на клас.

По аналогия с обикновените свойства (виж Свойства), вместо да се укаже *статичен клас-метод*, може да се укаже и име на поле. То също трябва да бъде *поле на клас*.

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

type
  TMyClass = class
  strict private
    // Alternative:
    // FMyProperty: Integer; static;
    class var
      FMyProperty: Integer;
    class procedure SetMyProperty(const Value: Integer); static;
  public
    class property MyProperty: Integer
      read FMyProperty write SetMyProperty;
  end;

class procedure TMyClass.SetMyProperty(const Value: Integer);
begin
  Writeln('MyProperty changes!');
  FMyProperty := Value;
end;

begin
  TMyClass.MyProperty := 123;
  Writeln('TMyClass.MyProperty is now ', TMyClass.MyProperty);
end.
```

9.7. Помощници за клас

Методът е просто процедура или функция вътре в класа. Извън класа го извиквате със специален синтаксис `MyInstance.MyMethod(⋯)`. След известно време привиквате да мислите, че *ако искам да извърша действие Action с инстанция X, пиша `X.Action(...)`*.

Но понякога трябва да кодирате нещо, което по смисъла си е *действие върху инстанция от клас TMyClass*, но без да модифицирате изходния код на *TMyClass*. Понякога това е така, защото изходния код не е ваш и не искате да го променяте. Понякога това се дължи на някакви зависимости — добавянето на нов метод като `Render` към клас `TMy3DObject` изглежда проста идея, но може би базовата реализация на класа `TMy3DObject` трябва да се поддържа независима от кода за изобразяване? Би било по-добре да "подобрите" съществуващ клас и

да добавите функционалност към него, без да променяте изходния му код.

Простия начин да го направите е да създадете глобална процедура, която приема екземпляр на `TMy3DObject` като свой първи параметър.

```
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;
```

Това работи идеално, но недостатъкът е, че извикването изглежда малко грозно. Докато обикновено извиквате действия като `X.Action(⋯)`, в този случай трябва да ги извиквате като `Render(X, ⋯)`. Би било добре да можете просто да напишете `X.Render(⋯)`, дори когато `Render` не е имплементирано в същия модул като `TMy3DObject`.

За това са пригодени *помощниците за клас*. Те са просто начин за прилагане на процедури / функции, които работят върху даден клас и които се извикват като нормални методи, но всъщност не са такива - те са добавени отвън към дефиницията на `TMy3DObject`.

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

procedure TMy3DObjectHelper.Render(const Color: TColor);
var
  I: Integer;
begin
  { забележете, че тук достъпваме ShapesCount и Shape без да ги квалифицираме }
  for I := 0 to ShapesCount - 1 do
    RenderMesh(Shape[I].Mesh, Color);
end;
```

ЗАБЕЛЕЖКА

По-общото понятие е *"Помощник за тип"*. Чрез тях можете да добавяте методи дори към примитивни типове, като цели числа или enum. Можете също да добавите *"помощници за запис"* към (познахте...) записи. Вижте http://lists.freepascal.org/fpc-announce/2013-February/000587.html .

9.8. Виртуални конструктори, деструктори

Името на деструктора е винаги `Destroy`, той е виртуален (защото трябва да се извика по време на изпълнение без да е известен точния клас) и е без параметри.

По конвенция името на конструктора е `Create`.

Можете да промените това име, но бъдете внимателни — ако дефинирате `CreateMy`, винаги предефинирайте `Create`, в противен случай потребителят все още може да извика `Create` на предшественика, заобикаляйки по този начин вашия `CreateMy` конструктор.

В `TObject` той не е виртуален и когато създавате наследници, можете свободно да променяте параметрите му. Новият конструктор ще скрие конструктора в предшественика (забележка: не поставяйте `overload`, освен ако не искате да се счупи).

В наследниците на `TComponent` трябва да замените неговия `constructor Create(AOwner: TComponent);`. При сериализацията, за да създадете клас, без да знаете неговия тип по време на компилиране, наличието на виртуални конструктори е много полезно (виж Препратки към клас по-горе).

9.9. Изключение в конструктор

Какво се случва, ако възникне изключение по време на изпълнението на конструктор? Редът:

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

в този случай не се изпълнява докрай, на `X` не може да се присвои стойност ... кой тогава ще почисти полусъздадената инстанция?

Решението в Object Pascal е, че в случай, че възникне изключение в рамките на конструктор, тогава се извиква деструкторът. Това е причина, поради която *вашият деструктор трябва да е стабилен*, т.е. трябва да работи при всякакви обстоятелства, дори на полусъздадена инстанция на клас. Обикновено това е лесно, ако освобождавате всичко безопасно, като например чрез `FreeAndNil`.

Ние също трябва да разчитаме в такива случаи, че *паметта на класа е гарантирано нулирана точно преди кодът на конструктора да бъде изпълнен*. Знаем, че в началото всички препратки към клас са `nil`, всички цели числа са `0` и така нататък.

Така че долното ще работи без изтичане на памет:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

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('Предизвикано изключение от конструктор!');
  Gun2 := TGun.Create;
end;

destructor TPlayer.Destroy;
begin
  { в случай, че конструктора крашне, бихме могли
    да имаме ситуация с Gun1 <> nil и Gun2 = nil. Справете се с това.
    ... Всъщност в случая FreeAndNil ще се справи без
    допълнителни усилия от наша страна, защото FreeAndNil проверява
    дали инстанцията е nil преди да извика деструктора. }
  FreeAndNil(Gun1);
  FreeAndNil(Gun2);
  inherited;
end;

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

10. Интерфейси

10.1. Голи (CORBA) интерфейси

Интерфейсът декларира набор от методи (API[7]), по подобие на клас, но не дефинира тяхната реализация. Даден клас може да бъде наследник само на един предшестващ клас, но пък може да имплементира много интерфейси.

Може да преобразувате типово клас до всеки от интерфейсите, които той имплементира и после да *извикате методите през този интерфейс*. Това позволява по еднакъв начин да третирате класове, които не произлизат един от друг, но все пак имат някаква обща функционалност. Това е алтернативно решение на множественото наследяване в езика C++.

CORBA интерфейсите в Обектния Паскал действат много подобно на интерфейсите в Java

(https://docs.oracle.com/javase/tutorial/java/concepts/interface.html) или C# (https://msdn.microsoft.com/en-us/library/ms173156.aspx).

```
{$ifdef FPC}
  {$mode objfpc}{$H+}{$J-}
  {$interfaces corba} // See below why we recommend CORBA interfaces
{$else}
  {$message warn 'Delphi does not support CORBA interfaces, only COM, that change how
memory is managed. This example is not valid in Delphi.'}
  begin end.
{$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

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('Shooting... ');
  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);
    // The "C3 is IMyInterface" below is false,
    // so "UseThroughInterface(C3 as IMyInterface)" will not execute.
    if C3 is IMyInterface then
      UseThroughInterface(C3 as IMyInterface);
  finally
    FreeAndNil(C1);
    FreeAndNil(C2);
    FreeAndNil(C3);
  end;
end.
```

10.2. Интерфейси CORBA и COM

Защо представените по-горе интерфейси са наречени "CORBA"?

Името **CORBA** е неудачно. По-добро име би било **голи интерфейси**. Тези интерфейси са *"`изцяло езикова функционалност`"*. Използвайте ги когато искате да приравните различни класове, но искате де да споделят едно и също API.

Въпреки, че тези интерфейси могат да се използват заедно с *технологията CORBA (Common Object Request Broker Architecture)* (see https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture), те *не са* свързани по никакъв друг начин с нея.

Необходима ли е директивата `{$interfaces corba}`?

Необходима е, защото иначе се създават *COM интерфейси*. Това може да се укаже изрично с `{$interfaces com}`, но обикновено не е необходимо защото това е направено по подразбиране.

Не препоръчвам да се използват *COM интерфейси*, особено ако търсите нещо еквивалентно като в други езици. *CORBA интерфейсите* в Паскал са точно каквото бихте очаквали от интерфейсите в C# или Java. *COM интерфейсите* от друга страна имат

допълнителни възможности, които вероятно не бихте желали в случая.

Забележете, че директивата `{$interfaces xxx}` се отразява само на интерфейсите, които нямат предшественик (само с ключовата дума `interface` а не `interface(ISomeAncestor)`, т.е. не са наследили друг интерфейс) Ако интерфейса е наследник на друг интерфейс, той ще бъде от същия тип като предшественика си, независимо от директивата `{$interfaces xxx}`.

Какво е COM интерфейс?

COM интерфейс представлява _интерфейс наследяващ специалния интерфейс `IUnknown` _. Наследяването на `IUnknown`:

- Изисква вашите класове да дефинират методите `_AddRef` и `_ReleaseRef`. Правилното имплементиране на тези методи може да управлява жизнения цикъл на вашите обекти с помощта на броене на препратки (reference-counting).
- Добавя метода `QueryInterface`.
- Позволява взаимодействие с *технологията COM (Component Object Model)*.

Защо не препоръчвам използването на COM интерфейси?

Тъй като *COM интерфейсите* "съвместяват" две функции, които според мен не бива да са свързани (а "ортогонални"): *множествено наследяване* и *броене на препратки*. Други езици за програмиране използват отделни механизми за тези две функции.

За да бъде ясно: **reference-counting**, което служи за автоматично управление на паметта (в прости ситуации и без цикли), е **много полезна функция**. Но **обвързването и с интерфейсите (вместо да се реализират ортогонално) в моите очи е много неподходящо**. Определено не отговаря на моята практика.

- Понякога ми е нужно да преобразувам някои от моите (несвързани един с друг) класове към общ интерфейс.
- Понякога ми е нужно автоматично да освобождавам паметта заета от обектите с помощта на броене на препратки.
- *Може би* някой ден ще ми се прииска да използвам *технологията COM*.

Но това са различни и несвързани изисквания. Съвместяването им в едно по мое мнение е контра-продуктивно, защото създава следните проблеми:

- Ако искам да *преобразувам класове към общ API интерфейс*, но не искам автоматично да освобождавам паметта на обектите с помощта на броене на препратки (искам да го правя ръчно), тогава *COM интерфейсите* са проблем. Дори броенето на препратки да се забрани със специални `_AddRef` и `_ReleaseRef` реализации, все пак трябва да внимавате никога да не виси временна препратка към интерфейс, след като сте освободили екземпляра на класа. Повече подробности за това в следващия раздел.
- Ако искам да имам *броене на препратки*, но нямам нужда от допълнително API към това на класа, тогава трябва да изкопирам декларациите на методи в интерфейси, т.е. да направя по един интерфейс за всеки клас. Това е е контра-продуктивно. Бих предпочел да имам *умни указатели* (smart pointers) като отделна езикова функция, която да не е обвързана с интерфейси (тя за щастие идва:).

Ето защо съветвам да използвате *CORBA интерфейси* и директивата `{$interfaces corba}` във всички съвременни кодове, които използват интерфейси.

Delphi засега има само COM интерфейси, така че трябва да използвате COM интерфейси, ако вашият код трябва да е съвместим с Delphi.

Можем ли да имаме броене на препратки с интерфейси CORBA?

Да. Просто добавете методи `_AddRef` / `_ReleaseRef`. Няма нужда да се наследява `IUnknown`. Въпреки че в повечето случаи, ако искате броене на препратки с вашите интерфейси, можете просто да използвате COM интерфейси.

10.3. Интерфейсни GUIDs

GUID са привидно произволни символни поредици `['{ABCD1234-⋯}']`, които виждате поставени във всяка дефиниция на интерфейс. Да, те са случайни и за съжаление са необходими.

GUID са без значение, ако не планирате да се интегрирате с технологии като *COM* или *CORBA*. Но те са необходими за правилното изпълнение. Не се заблуждавайте от компилатора, който за съжаление ви позволява да декларирате интерфейси без GUID.

Без (уникалните) GUID, вашите интерфейси ще бъдат третирани еднакво от оператора `is`. В действителност, той ще върне `true`, ако вашият клас поддържа *който и да е* от вашите интерфейси. Магическата функция `Supports(ObjectInstance, IMyInterface)` се държи малко по-добре, тъй като отказва да бъде компилирана за интерфейси без GUID. Това важи както за интерфейсите CORBA, така и за COM, от FPC 3.0.0.

Така че, за да сте сигурни, винаги трябва да декларирате GUID за вашия интерфейс. Можете да използвате *Lazarus* генератора на GUID (натиснете `Ctrl + Shift + G` в редактора). Или можете да използвате онлайн услуга като https://www.guidgenerator.com/ .

Или можете да напишете свой собствен инструмент за това, като използвате функциите `CreateGUID` и `GUIDToString` в RTL. Вижте примера по-долу:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

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

10.4. Интерфейси с броене на препратки (COM)

COM интерфейсите добавят две допълнителни функции:

1. интеграция с COM (технология от Windows, достъпна и на Unix чрез *XPCOM*, използвана от Mozilla),
2. броене на препратки (което води до автоматично унищожаване, когато всички препратки към интерфейса излязат от обхват).

Когато използвате *COM интерфейси*, трябва да сте наясно с техния механизъм за *автоматично унищожаване* и връзката им с COM технологията.

На практика това означава, че:

- Вашият клас трябва да имплементира магическите методи `_AddRef`, `_Release` и `QueryInterface`. Или да наследи нещо, което вече ги е имплементирало. Конкретно изпълнение на тези методи може на практика да активира или деактивира функцията *reference-counting* на COM интерфейсите (въпреки че деактивирането й е донякъде опасно - вижте следващата точка).
  - Стандартният клас `TInterfacedObject` имплементира тези методи за да *разреши* преброяването на препратки.
  - Стандартният клас `TComponent` имплементира тези методи за да *забрани* преброяването на препратки. В **Castle Game Engine** ние ви даваме допълнителните полезни класове за наследяване `TNonRefCountedInterfacedObject` и `TNonRefCountedInterfacedPersistent` за тази цел, вижте https://github.com/castle-engine/castle-engine/blob/0519585abc13e8386cdae5f7dfef6f9659dc9b57/src/base/castleinterfaces.pas .
- Трябва да внимавате да не освобождавате класа, когато той може да бъде сочен от някои интерфейсни променливи. Понеже интерфейсът се освобождава с помощта на виртуален метод (тъй като *може да бъде reference-counted, дори и при хакнат метод _AddRef за да не се брои...*), не можете да освободите основния екземпляр на обекта, докато някаква интерфейсна променлива сочи към него. Вижте *"7.7 Броене на препратки"* в ръководството на FPC (http://freepascal.org/docs-html/ref/refse47.html).

Най-безопасният подход за използване на *COM интерфейси* е:

- да приемете факта, че са reference-counted,
- да наследите подходящите класове от `TInterfacedObject`,
- и да избягвате използването на истинския екземпляра на класа, вместо това винаги осъществявайте достъп до екземпляра през интерфейс, оставяйки броенето на референции да извърши освобождаването.

Това е пример за използване на такъв интерфейс:

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

```
  {$interfaces com}
{$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

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('Shooting... ');
  I.Shoot;
end;

var
  C1: IMyInterface;  // COM се грижи за унищожаването
  C2: IMyInterface;  // COM се грижи за унищожаването
  C3: TMyClass3;     // ВИЕ трябва да се погрижите за унищожаването
begin
```

```
  C1 := TMyClass1.Create as IMyInterface;
  C2 := TMyClass2.Create as IMyInterface;
  C3 := TMyClass3.Create;
  try
    UseThroughInterface(C1); // няма нужда от оператор "as"
    UseThroughInterface(C2);
    if C3 is IMyInterface then
      UseThroughInterface(C3 as IMyInterface); // това няма да се изпълни
  finally
    { Променливи C1 и C2 излизат от обхват и тук би трябвало да се
      унищожат автоматично.

      За разлика от тях, C3 е инстанция, която не се управлява от интерфейс
      и трябва да се унищожи ръчно. }
    FreeAndNil(C3);
  end;
end.
```

10.5. Използване на COM интерфейси с изключено броене

Както бе споменато в предишния раздел, вашият клас може да произхожда от `TComponent` (или подобен клас като `TNonRefCountedInterfacedObject` и `TNonRefCountedInterfacedPersistent`), който деактивира броенето на препратки за COM интерфейси. Това ви позволява да използвате тези интерфейси и въпреки това да освободите екземпляра на класа ръчно.

Трябва да внимавате в този случай да не освободите екземпляра на класа, когато някаква интерфейсна променлива сочи към него. Запомнете, че всеки typecast `Cx as IMyInterface` също създава временна интерфейсна променлива, която може да присъства дори до края на текущата процедура. Поради тази причина примерът по-долу използва процедура `UseInterfaces` и освобождава екземплярите на класа *извън* на тази процедура (когато можем да сме сигурни, че временните интерфейсни променливи са извън обхвата).

За да избегнете тази бъркотия, обикновено е по-добре да използвате CORBA интерфейси, ако не е нужно да броите препратки.

```
{$ifdef FPC}
  {$mode objfpc}{$H+}{$J-}
  {$interfaces com}
{$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

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('Shooting... ');
  I.Shoot;
end;

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

procedure UseInterfaces;
begin
  // In FPC, you could also check using "is", like:
  //if C1 is IMyInterface then ...

  if Supports(C1, IMyInterface) then
    UseThroughInterface(C1 as IMyInterface);
  if Supports(C2, IMyInterface) then
    UseThroughInterface(C2 as IMyInterface);
  if Supports(C3, 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.
```

10.6. Преобразуване на интерфейси

Този раздел се отнася както за интерфейсите *CORBA*, така и за *COM* (все пак има някои изрични изключения за CORBA).

1. Прехвърлянето към тип интерфейс с помощта на оператора `as` прави проверка по време на изпълнение. Разгледайте следния код:

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

   Работи за всички случаи на `C1`, `C2`, `C3` в примерите в предишните раздели. Ако се изпълни, това ще доведе до грешка по време на изпълнение в случая на `C3`, който не имплементира `IMyInterface`.

   Използването на оператор `as` работи правилно, независимо дали `Cx` е деклариран като екземпляр на клас (като `TMyClass2`) или интерфейс (като `IMyInterface2`).

   Това обаче не е разрешено за CORBA интерфейси.

2. Вместо това можете изрично да конвертирате екземпляра до интерфейс:

   ```
   UseThroughInterface(Cx);
   ```

   В този случай конверсията трябва да е валидна по време на компилация. Така че това ще се компилира за `C1` и `C2` (които са декларирани като класове, които имплементират `IMyInterface`). Но няма да се компилира за `C3`.

   По същество тази конверсия изглежда и работи точно както и за обикновени класове. Където и да е необходим екземпляр на клас `TMyClass`, винаги можете да използвате там променлива, която е декларирана с клас на `TMyClass`, **или `TMyClass` потомък**. Същото правило важи и за интерфейсите. Няма нужда от изрично преобразуване на типа в

такива ситуации.

3. Можете също така да използвате `IMyInterface(Cx)`:

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

   Обикновено такъв синтаксис за преобразуване на типове показва *опасно, непроверено* преобразуване на типове. Ще се случат лоши неща, ако конвертирате към неправилен интерфейс. И това е вярно, ако преобразувате *клас към клас* или *интерфейс към интерфейс*, използвайки този синтаксис.

   Тук има малко изключение: ако `Cx` е деклариран като клас (като `TMyClass2`), тогава това е тип, който трябва да е валиден по време на компилация. Така че прехвърлянето на *на клас към интерфейс* по този начин е безопасно, бързо (проверено по време на компилиране) преобразуване на типа.

За да тествате всичко това, поиграйте си с този примерен код:

```
{$ifdef FPC} {$mode objfpc}{$H+}{$J-} {$endif}
{$ifdef MSWINDOWS} {$apptype CONSOLE} {$endif}

// {$interfaces corba} // забележете, че "as" конверсии за CORBA няма да се компилират

uses Classes;

type
  IMyInterface = interface
  ['{7FC754BC-9CA7-4399-B947-D37DD30BA90A}']
    procedure One;
  end;

  IMyInterface2 = interface(IMyInterface)
  ['{A72B7008-3F90-45C1-8F4C-E77C4302AA3E}']
    procedure Two;
  end;

  IMyInterface3 = interface(IMyInterface2)
  ['{924BFB98-B049-4945-AF17-1DB08DB1C0C5}']
    procedure Three;
  end;

  TMyClass = class(TComponent, IMyInterface)
    procedure One;
  end;

  TMyClass2 = class(TMyClass, IMyInterface, IMyInterface2)
    procedure One;
    procedure Two;
  end;
```

```
procedure TMyClass.One;
begin
  Writeln('TMyClass.One');
end;

procedure TMyClass2.One;
begin
  Writeln('TMyClass2.One');
end;

procedure TMyClass2.Two;
begin
  Writeln('TMyClass2.Two');
end;

procedure UseInterface2(const I: IMyInterface2);
begin
  I.One;
  I.Two;
end;

procedure UseInterface3(const I: IMyInterface3);
begin
  I.One;
  I.Two;
  I.Three;
end;

var
  My: IMyInterface;
  MyClass: TMyClass;
begin
  My := TMyClass2.Create(nil);
  MyClass := TMyClass2.Create(nil);

  // Това не може да с компилира, не е известно дали My е IMyInterface2.
  // UseInterface2(My);
  // UseInterface2(MyClass);

  // Това се компилира и работи.
  UseInterface2(IMyInterface2(My));
  // Това не може да с компилира. Преобразуването InterfaceType(ClassType) се
проверява при компилация.
  // UseInterface2(IMyInterface2(MyClass));

  // Това се компилира и работи.
  UseInterface2(My as IMyInterface2);
  // Това се компилира и работи.
  UseInterface2(MyClass as IMyInterface2);
```

```
  // Това се компилира но не работи при изпълнение, с грозно "Access violation".
  // UseInterface3(IMyInterface3(My));
  // Това не може да с компилира. Преобразуването InterfaceType(ClassType) се
проверява при компилация.
  // UseInterface3(IMyInterface3(MyClass));

  // Това се компилира но не работи при изпълнение, с хубаво "EInvalidCast: Invalid
type cast".
  // UseInterface3(My as IMyInterface3);
  // Това се компилира но не работи при изпълнение, с хубаво "EInvalidCast: Invalid
type cast".
  // UseInterface3(MyClass as IMyInterface3);

  Writeln('Край');
end.
```

11. Относно този документ

Copyright Michalis Kamburelis.

Изходният код на този документ е във формат AsciiDoc на https://github.com/modern-pascal/modern-pascal-introduction. Предложения за корекции и допълнения, кръпки и заявки за изтегляне са винаги добре дошли:) Можете да се свържете с мен чрез GitHub или да изпратите имейл на michalis@castle-engine.io. Моята WEB страница е https://michalis.xyz/. Този документ е свързан в секция *Documentation* на *Castle Game Engine* website https://castle-engine.io/.

Можете да разпространявате и дори да променяте този документ свободно, под същите лицензи като Wikipedia https://en.wikipedia.org/wiki/Wikipedia:Copyrights :

- *Creative Commons Attribution-ShareAlike 3.0 Unported License (CC BY-SA)*
- or the *GNU Free Documentation License (GFDL) (unversioned, with no invariant sections, front-cover texts, or back-cover texts)* .

Thank you for reading!

Превод на Български език: Юлиян Иванов, 2023

[1] модул = Unit

[2] генерици = Generics

[3] интерфейс = Interface

[4] "опаковъчни" функции = wrappers

[5] речник = Dictionary, a.k.a. Associative array

[6] приятели = friends

[7] API = Application Program Interface