

给程序员的

现代 Object Pascal

介绍

作者: Michalis Kamburelis <https://castle-engine.io>

翻译: robsean

翻译时间: 2018 年 09 月 27 日---2019 年 07 月 02 日 www.lazaruscn.top

目录

给程序员的 Modern Object Pascal 介绍	1
1. 为什么	4
2. 基础	5
2.1. "Hello world"程序	5
2.2. 函数, procedures (过程), primitive (原始) 类型	5
2.3. 测试(if)	7
2.4. Logical (逻辑), relational (关系式) 和 bit-wise (位-判断) 运算符	8
2.5. 对多个值 (case) 测试单个表达式	8
2.6. Enumerated (枚举) 和 ordinal types (有序类型) 和 sets (集合) 和 constant-length arrays (定长数组)	9
2.7. 循环(for, while, repeat, for .. in)	10
2.8. 输出, 记录	12
2.9. 转换为字符串	13
3. 单元	14
3.1. 单元相互使用	14
3.2. 使用单元名称限定(Qualifying)标识符	15
3.3. 从另一个单元接触(expose)一个单元标识符	18
4. 类	19
4.1. 基础	19
4.2. 继承 (Inheritance), is, as	19
4.3. 属性	21
4.4. 异常	23
4.5. Visibility specifiers (可视化说明符/分类符)	24
4.6. 默认 原型(ancestor)	24
4.7. Self	24
4.8. 调用继承的方法(inherited method)	25
4.9. 虚拟 (Virtual) 方法, override (重写) 和 reintroduce (再引入)	27
5. 释放类	31
5.1. 记住释放类实例	31
5.2. 如何释放	31
5.3. 手动和自动释放	31
5.4. 虚拟析构函数(destructor)被称为 Destroy(销毁)	34
5.5. 释放 notification(通知)	35
6. 运行时库	38
6.1. 使用 streams (流) 输入/输出	38
6.2. 容器 (lists(列表), dictionaries(词典))使用泛型(generics)	38
6.3. Cloning (克隆) : TPersistent.Assign	44
7. 各种各样的语言特征	48
7.1. 局部(嵌套)例行程序 (routines)	48
7.2. Callbacks (回调) (亦称 events (事件), 亦称 pointers to functions (指针到函数), 亦称 procedural variables (过程变量))	49
7.3. 泛型(Generics)	51
7.4. Overloading (重载)	53

7.5. 预处理程序.....	53
7.6. 记录.....	55
7.7. 旧样式对象.....	56
7.8. 指针.....	56
7.9. 运算符(Operator)重载.....	57
8. 高级的类特征.....	60
8.1. private（私有的）和 strict private（绝对的私有的）	60
8.2. 在类和嵌套的(nested)类的内部更多原料(stuff).....	60
8.3. 类方法.....	61
8.4. 类引用（ <i>Class reference</i> ）	61
8.5. 静态类方法.....	64
8.6. 类属性和变量.....	65
8.7. 类助手（helpers）	66
8.8. 虚拟构造函数，析构函数.....	67
8.9. 在构造函数中的一个异常.....	67
9. 接口.....	69
9.1. Bare (CORBA) 接口.....	69
9.2. CORBA 和接口的 COM 类型	70
9.3. 接口 GUIDs（全球唯一的标志符）	72
9.4. 引用计数(COM)接口.....	73
9.5. 禁用引用计数使用 COM 接口.....	75
9.6. 不带有"as"操作符类型强制转换接口	77
10. 关于这个文档.....	78

1. 为什么

在这里外有很多关于 Pascal 的书和资源，但是太多的它们讨论老式的(old) Pascal，没有 classes（类），units（单元）或者 generics（泛型）。

所以我写这个快速介绍，什么是我称为 **modern Object Pascal（现代 Object Pascal）**。大多数的程序员使用不真正称为 "*modern Object Pascal*" 的 Pascal，我们仅仅是称它为 "*our Pascal*".。但是当介绍语言时，我感觉强调它是一个现代的，面向对象的语言是重要的。自从很久以前很多人在学校学习的老式的 (Turbo) Pascal 以来，它发展了很多。充满智慧的-特征 (Feature-wise)，它非常类似于 C++ 或 Java 或 C#。

- 它有你要求的所有现代的特征 — classes（类），units（单元），interfaces（接口），generics（泛型）...
- 它编译到快速的，原生的代码，
- 它非常类型（type）安全，
- 高级，但是如果你需要也能是低级。

它也有极好的，便携式的和开放-源文件的编译器，被称为 *Free Pascal Compiler*，<http://freepascal.org/>。和一个附随的 IDE (编译器，调试器，一个可视化组件的库，窗体设计器)，被称为 *Lazarus* <http://lazarus.freepascal.org/>。我自己，我是 *Castle Game Engine* 的创建者，<https://castle-engine.io/>，它是一个绝妙的便携式的 3D 和 2D 游戏引擎，使用这个语言来在很多平台(Windows, Linux, MacOSX, Android, iOS, web 插件.上创建游戏。

这个介绍是主要针对已经在其他语言中有经验的程序员。我们将不在这里涉及一般的概念，像“什么是一个类”，我们将仅展示如何在 Pascal 中完成它们。

2. 基础

2.1. "Hello world"程序

```
{ $mode objfpc } { $H+ } { $J- } // Just use this line in all modern sources
```

```
program MyProgram; // Save this file as myprogram.lpr
begin
  Writeln('Hello world!');
end.
```

这是一个完整的程序，你可以编译和运行。

- 如果你使用命令行 FPC，只创建一个新的文件 `myprogram.lpr`，并执行 `fpc myprogram.lpr`。
- 如果你使用 Lazarus，创建一个新的工程（菜单 工程→新建工程→简单程序）。保持它为 `myprogram` 并粘贴这些源文件代码为主文件（main file）。使用菜单项目编译运行→编译。
- 这是一个命令程序，所以在任何一种情况下 — 只从命令行运行编译的命令。

这篇文章的剩余部分讨论对象 Pascal 语言，所以不要期望看到比命令行资料更有趣的任何事。如果你想看到绝妙的一些事，在 *Lazarus*（工程→新建工程→应用程序）中，创建一个新的 GUI 工程。Voila — 一个工作的 GUI 应用程序，跨平台，带有任何地方的原生外观，使用一个合适的可视化组件库。*Lazarus* 和 *Free Pascal Compiler* 带来很多准备的用于网络，GUI，数据库，文件格式（XML，json，images...），线程和你可能需要的其他每种事的单元。我也提及我的绝妙的 *Castle Game Engine* :)

2.2. 函数，procedures（过程），primitive（原始）类型

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

program MyProgram;

procedure MyProcedure(const A: Integer);
begin
  Writeln('A + 10 is: ', A + 10);
end;

function MyFunction(const S: string): string;
begin
  Result := S + 'strings are automatically managed';
end;

var
  X: Single;
begin
```

```

Writeln(MyFunction('Note: '));
MyProcedure(5);

// Division using "/" always makes float result, use "div" for integer division
X := 15 / 5;
Writeln('X is now: ', X); // scientific notation
Writeln('X is now: ', X:1:2); // 2 decimal places
end.

```

为从一个函数返回一个值，分配一些事到不可思议的 **Result** 变量。你可以自由地读和设置 **Result**，像一个局部变量。

```

function MyFunction(const S: string): string;
begin
    Result := S + 'something';
    Result := Result + ' something more!';
    Result := Result + ' and more!';
end;

```

你也可以对待函数名称（像在上面的示例中的 **MyFunction**）作为变量，你可以分配这些。不过我将在新的代码中 **discourage**（阻止）它，因为当使用在赋值表达式的右侧时，它看起来“fishy（值得怀疑的）”。当你想读或设置函数结果时，也仅仅使用 **Result**。

如果你想递归地调用函数自身，你当然可以这么做。如果你正在递归地调用一个不带参数的函数，务必指定 **parenthesis**（圆括号）（即使在 **Pascal** 中，你可以经常对一个不带参数的函数省略 **parentheses**（圆括号）），这使得一个递归地调用一个不带参数的函数，不同于访问这个函数的当前结果。像这个：

```

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

```

你可以在它到达最后的 **end;** 前，调用 **Exit** 到 **procedure**（过程）或函数执行的 **end**。如果你在一个函数中调用不带参数的 **Exit**。它将返回你设置的最后的事作为结果。你也可以使用 **Exit(X)** 结构体（construct），来设置函数结果并立即退出—这像在 C 类语音中的 **return X** 结构体（construct）。

```

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

```

2.3. 测试(if)

当一些条件满足（satisfied）时，使用 if .. then 或 if .. then .. else 来运行一些代码。不像在 C 类语言，在 Pascal 中，你不用在括号（ parenthesis）中包裹条件。

```
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;

  // equivalent to above
  B := A > 10;
  if B then
    DoSomething
  else
    DoSomethingElse;
end;
```

else 是与最近的 if 成对的。所以这个工作正如你预期：

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

虽然上面带嵌套的 if 的示例是正确的，在这种情况下，它最好经常放置嵌套的 if 在一个 begin ... end 语句块内部。这使得代码更易理解的阅读，并且即使你弄乱缩进，它将保持易理解。示例的改进版在下面。当你在下面的代码中添加或移除一些 else 分句，哪个条件（到 A 测试或 B 测试）将应用是很明显的，所以它更不易错。

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

```
end;
```

2.4. Logical (逻辑), relational (关系式) 和 bit-wise (位-判断) 运算符

logical operators (逻辑运算符) 被称为 `and`, `or`, `not`, `xor`。它们的意思是显而易见的 (搜索 "exclusive or", 如果你不确认什么是 `xor`:)。它们接受 *boolean arguments* (布尔值参数), 并返回一个 *boolean* (布尔值)。当两个参数是整型值时, 它们也可以充当 *bit-wise operators* (位-判断运算符), 在这种情况下它们将返回一个整型数。

relational (comparison) (关系式/ (比较)) 运算符是 `=`, `<>`, `>`, `<`, `<=`, `>=`。如果你习惯于 C 类语言, 注意在 Pascal 中, 你比较两个值 (检查它们相等), 使用一个单一的等于符号 `A = B` (不像在 C 中, 在那里你使用 `A == B`)。在 Pascal 中特殊的 *assignment* (赋值) 运算符是 `:=`。

logical (逻辑) (或 *bit-wise* (位-判断)) 运算符有一个比 *relational* (关系式) 运算符更高的优先权。所以, 你可能在一些表达式周围需要使用圆括号。

例如, 这是一个编译错误:

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

上面未能够来编译, 因为编译器看到 *bit-wise* (位-判断) 和在内部的: (0 和 B)。

这是正确的:

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

short-circuit evaluation (短路求值) 被使用。考虑这个表达式:

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

- 它保证 `MyFunction(X)` 将被首先被求数值 (evaluated)。
- 如果 `MyFunction(X)` 返回 `false`, 那么表达式的值是已知的 (虚拟 (false) 的值, 无论什么也是虚拟 (false) 的), 并且 `MyOtherFunction(Y)` 将根本不执行。
- 相似的规则用于 `or` 运算符。在那里, 如果表达式是已知的将是 `true` (因为第 1 个运算数是 `true`), 第 2 个运算数不被求数值 (evaluated)。
- 当写表达式时, 这是特别的有用的, 像:

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

这将工作完好, 即使当 A 是 `nil` (空)。

2.5. 对多个值 (case) 测试单个表达式

如果一个不同的动作依赖一些表达式的值将会被执行, 那么 `case .. of .. end` 声明是有用的。

```
case SomeValue of
  0: DoSomething;
  1: DoSomethingElse;
  2: begin
    IfItsTwoThenDoThis;
    AndAlsoDoThis;
```



```
end;
3..10: DoSomethingInCaseItsInThisRange;
11, 21, 31: AndDoSomethingForTheseSpecialValues;
else DoSomethingInCaseOfUnexpectedValue;
end;
```

else 分句是可选的。当没有条件匹配，并且这里没有 else，那么没有事将发生。

在你来自的 C 类语言中，在这些语言中使用 switch 声明比较这些，你将注意到这里是没有自动的 *fall-through*。在 Pascal 中这是一个深思熟虑的好事（deliberate blessing）。你没有必要记住放置 break 指令（instructions）。在每一次执行中，在 case 的最多一个分支被执行，就是这样。

2.6. Enumerated（枚举）和 ordinal types（有序类型）和 sets（集合）和

constant-length arrays（定长数组）

在 Pascal 中枚举类型是一个非常好的，难懂的（opaque）类型。你可能将在其他语言中更加经常地使用它：)

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

惯例是使用类型名称的两个字母的简写来前置枚举名称，因此 `ak` = "*Animal Kind*"的简写。这是一个有用的惯例，因为枚举名称是在单元（全局）命名空间中。所以使用 `ak` 前缀来在它们前加前缀，你使用其他标识符降低冲突的风险。

在名称中的冲突不是一个显示堵塞物（show-stopper）。对不同的单元来定义相同的标识符是可以的。但是尝试无论如何避免冲突是一个好注意，保持代码简单带理解和检索目标行命令（grep）。

你可以通过指令 `{ $scopedenums on }` 避免在全局命名空间中放置枚举名称。这意味着你将不得不通过一个类型的名称部分成功访问它们，像 `TAnimalKind.akDuck.`。在这种情况下，`ak` 需要的前缀消失，并且你将可能调用枚举 `Duck, Cat, Dog`。这类似于 C# 枚举。

事实上，枚举类型是难懂的（opaque），意味着它不能仅仅被分配或来自一个整形数。然而，对于特殊使用，你可以使用 `Ord(MyAnimalKind)` 来强制地转换 `enum` 到 `int`，或者 `typecast TAnimalKind(MyInteger)` 来强制地转换 `int` 到 `enum`。在后面的实例中，确保来首先检查是否 `MyInteger` 在良好的范围内 (`0 to Ord(High(TAnimalKind))`)。枚举和有序类型可以作为数组索引被使用：

```
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;
```

它们也能被用于创建集合（一个内部的位字段(bit-fields)）：

```

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. 循环(for, while, repeat, for .. in)

```

{$mode objfpc}{$H+}{$J-}
{$R+} // range checking on - nice for debugging
var
  MyArray: array [0..9] of Integer;
  I: Integer;
begin
  // initialize
  for I := 0 to 9 do
    MyArray[I] := I * I;

  // show
  for I := 0 to 9 do
    Writeln('Square is ', MyArray[I]);

  // does the same as above
  for I := Low(MyArray) to High(MyArray) do
    Writeln('Square is ', MyArray[I]);

  // does the same as above
  I := 0;
  while I < 10 do
  begin
    Writeln('Square is ', MyArray[I]);
    I := I + 1; // or "I += 1", or "Inc(I)"
  end;

  // does the same as above
  I := 0;
  repeat

```

```

    Writeln('Square is ', MyArray[I]);
    Inc(I);
until I = 10;

// does the same as above
for I in MyArray do
    Writeln('Square is ', I);
end.

```

关于 repeat 和 while 循环:

这两个循环之间有两个不同:

1. 循环条件有一个相反的意思。在 `while .. do` 中, 你将告诉它 *什么时候继续*。但是在 `repeat .. until` 中, 你将告诉它 *什么时候停止*。
2. 在 `repeat` 的情况下, *在开始时, 条件将不被检查*。所以 `repeat` 循环总是运行最少一次。

关于 for I := ... 循环:

`for I := .. to .. do ...` 结构类似于 C-like `for` 循环。然而, 它更多约束。因为你不能指定任意的 `actions/tests` 来控制循环迭代。对重复一个相邻数 (或其他有序类型) 是严格的。你仅有的灵活性是你可以使用 `downto` 代替 `to`, 使数字下降。

作为交换, 它看起来整洁, 并且在执行时非常地优化。特别地, 在循环开始前, 表达式的下界和上界仅被计算一次。

注意, 在循环完成后, 因为可能的优化, 循环计数器变量的值 (在这个示例中 `I`) 应该被考虑成 *undefined* (未定义的)。在循环后访问 `I` 的值可能导致一个编译器错误。除非你提前通过 `Break` 或 `Exit` 退出循环: 在这种情况下, 计数器变量被确定保持最后的值。

关于 for I in ... do 循环:

`for I in .. do ..` 类似于在很多现代语言中的 `foreach` 结构。它在很多内置的类型上智能地工作:

- 它能在数组中迭代(iterate)全部的值 (上面的示例)。
- 它能迭代(iterate)一个枚举类型的全部可能的值:

```

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

```

- 它能迭代(iterate)包含在集合中的全部项目:

```

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

```

- 它可以工作在自定义的 `list` 类型, 通用 (generic) 或否, 像 `TObjectList` 或 `TFPGObjectList`.

```

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

type
    TMyClass = class
        I, Square: Integer;

```

```

end;
TMyClassList = specialize TFPGObjectList<TMyClass>;

var
  List: TMyClassList;
  C: TMyClass;
  I: Integer;
begin
  List := TMyClassList.Create(true); // true = owns children
  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('Square of ', C.I, ' is ', C.Square);
      finally FreeAndNil(List) end;
    end.
end.

```

我们还没有解释类的概念，所以对你来说，最后的示例可能还不容易理解——只有继续进行，随后它将容易理解 :)

2.8. 输出，记录

为在 Pascal 中简单地输出字符串，使用常规的 `Write` 或 `Writeln`。后者自动在结尾添加一个新行。

在 Pascal 中这是一个"特殊能力(magic)"惯例。它取得参数的一个变量值，并且它们可以有任何类型。当显示时，它们全部转换到字符串，用一种特殊的的语法来具体说明衬语(padding)和数字准确度。

```

Writeln('Hello world!');
Writeln('You can output an integer: ', 3 * 4);
Writeln('You can pad an integer: ', 666:10);
Writeln('You can output a float: ', Pi:1:4);

```

为明确地在字符串中使用新行，使用 `LineEnding` 常量(来自 FPC RTL)。(*Castle Game Engine* 也定义一个简写的 `NL` 常量。) Pascal 字符串不解释任何特殊的反斜线符号系列， 所以写：

```
Writeln('One line.\nSecond line.');// INCORRECT example
```

不工作，像你们中的一些想的。这样将工作：

```
Writeln('One line.' + LineEnding + 'Second line.');
```

或仅仅这样：

```
Writeln('One line.');//
Writeln('Second line.');//

```

注意这将仅在 *控制台* 程序下工作。确保在你的主程序文件中有 `{ $apptype CONSOLE }` 定义(而

不是{\$apptype GUI})。在一些操作系统上，它实际上不重要，并且将总是工作(Unix)，但是在一些操作系统上尝试从一个 GUI 应用程序中尝试写一些东西是一个错误(Windows)。

在 Castle Game Engine 中: 使用 `WritelnLog` 或 `WritelnWarning`，从来不 `Writeln`，来打印调试信息。它们将总是指向一些有用的输出。在 Unix 上，标准输出。在 Windows GUI 应用程序上，日志文件。在 Android 上，*Android 日志记录设备* (可视，当你使用 `adb logcat` 时)。当你写一个命令行应用程序（像一个 3D 模型转换器/生产者），和你知道那个 *标准输出* 是可用的时，`Writeln` 的使用应该被局限于实例。

2.9. 转换为字符串

为转换参数的一个任意数到一个字符串（而不是仅直接输出它们），你有几个选项。

- 你可以转换特定的类型到字符串，使用专用的函数，像 `IntToStr` 和 `FloatToStr`。此外，你可以在 Pascal 中通过简单地添加它们连接字符串。所以，你可以创建一个字符串，像这： `'My int number is ' + IntToStr(MyInt) + ', and the value of Pi is ' + FloatToStr(Pi)`.
 - **优点：** 绝对地灵活。这里有很多 `XxxToStr` 重载(overloaded)版本和近亲(friends) (像 `FormatFloat`)，覆盖很多类型。
 - **另一个优点：** 与相反的函数并存。为转换一个字符串（例如，用户输入）回到一个整形数或浮点数，你使用 `StrToInt`，`StrToFloat` 和近亲(friends) (像 `StrToIntDef`)。
 - **缺点：** 大量的 `XxxToStr` 系列互相关联的事物调用和字符串看起来不漂亮。
- `Format` 函数，使用，像 `Format('%d %f %s', [MyInt, MyFloat, MyString])`。在 C 类应用中像 `sprintf` 函数。它插入参数到样本(pattern)中的占位符(placeholders)。占位符可以使用特殊的语法来支配格式化，例如 `%.4f` 导致在一个浮点格式中，在小数点后带有 4 个数字。
 - **优点：** 样本(pattern)字符串从参数(arguments)的分离看起来整洁。如果你需要更改样本(pattern)字符串，而不触及参数(arguments) (例如，当翻译时)，你可以容易地做它。
 - **另一个优点：** 没有编译器特殊能力(magic)。你可以在你自己的程序(routines)(声明参数为一个常量的数组(array of const))中使用相同的语法来传递参数的一个任意类型的很多的参数。你可以再向下传递这些参数到 `Format` 中，或者解构(deconstruct)参数的列表，并做任何你喜欢的事。
 - **缺点：** 编译器不能检查样品(pattern)是否匹配参数。使用一个错误的占位符类型将在运行时导致一个异常(`EConvertError` 异常，不是任何事严重的像一个记忆体区段错误(segmentation fault))。
- `WriteStr(TargetString, ...)` 程序(routines)行为很像 `Write(...)`，要求结果保存到 `TargetString`。
 - **优点：** 它支持 `Write` 的所有特征，包含特殊的语法格式像 `Pi:1:4`。
 - **缺点：** 特殊的语法格式是一个 "编译器特殊能力(magic)"，为程序 (routines) 这样特殊地实施(implemented)。有时很麻烦，例如，你不能创建你自己的程序 (routines) `MyStringFormatter(...)`，那可能允许特殊的语法像 `Pi:1:4`。鉴于这个原因 (也因为不能在大多数的 Pascal 编译器中长时间的实施)，这个构造 (construction) 不是很流行。

3. 单元

单元允许你组团通用材料(任何事，可以被声明), 为其他单元和程序使用。它们等于在其他语言中的模块 (*modules*) 和包 (*packages*)。它们有一个 **interface** (接口) 分句, 在这里, 你声明什么对其他单元和程序是可用的, 然后是 **implementation** (实施) 分句。保存 **MyUnit** 为 **myunit.pas** (小写.pas 扩展名)。

```
{ $mode objfpc } { $H+ } { $J- }  
unit MyUnit;  
interface  
  
procedure MyProcedure(const A: Integer);  
function MyFunction(const S: string): string;  
  
implementation  
  
procedure MyProcedure(const A: Integer);  
begin  
    Writeln('A + 10 is: ', A + 10);  
end;  
  
function MyFunction(const S: string): string;  
begin  
    Result := S + 'strings are automatically managed';  
end;  
  
end.
```

最终程序被保存为 **myprogram.lp** 文件(**lpr** = **Lazarus** 程序文件; 在 **Delphi** 中, 你将使用 **.dpr**)。注意其他的惯例可能在这里, 例如, 一些工程仅对主程序文件使用 **.pas**, 一些对单元或程序使用 **.pp**。我建议对单元使用 **.pas**, 对 **FPC/Lazarus** 程序使用 **.lpr**。一个程序可以通过一个 **uses** 关键字使用一个单元:

```
{ $mode objfpc } { $H+ } { $J- }  
  
program MyProgram;  
  
uses MyUnit;  
  
begin  
    Writeln(MyFunction('Note: '));  
    MyProcedure(5);  
end.
```

3.1. 单元相互使用

一个单元也能使用其他单元。另一个单元可以被使用在 **interface** (接口) 分句中, 或仅在

implementation（实施）分句中。前面的允许定义新的 public（公共的）材料/东西(stuff) (procedures(过程),类型...) 在其他的材料/东西(stuff)的上面。后面的被更多的限制(如果你仅在 implementation（实施）分句中使用一个单元，你仅能在你的 implementation（实施）中使用它的标识符(identifiers))。

```
{ $mode objfpc } { $H+ } { $J- }  
unit AnotherUnit;  
interface  
  
uses Classes;  
  
{ The "TComponent" type (class) is defined in the Classes unit.  
  That's why we had to use the Classes unit above. }  
procedure DoSomethingWithComponent(var C: TComponent);  
  
implementation  
  
uses SysUtils;  
  
procedure DoSomethingWithComponent(var C: TComponent);  
begin  
  { The FreeAndNil procedure is defined in the SysUtils unit.  
    Since we only refer to it's name in the implementation,  
    it was OK to use the SysUtils unit in the "implementation" section. }  
  FreeAndNil(C);  
end;  
  
end.
```

在 *interface*（接口）中不允许有循环（*circular*）单元依赖关系。换句话说，*interface*（接口）分句中在两个单元不能相互使用。原因是，为了“理解”一个单元的 *interface*（接口）部分，编译器必需首先“理解”在 *interface*（接口）部分中它使用的全部的单元。Pascal 语言严格遵循这个规则，并且它允许一个快速编译和在编译器端完全地自动检测：什么单元需要被重新编译。在 Pascal 中不需要为一个简单的编译任务使用复杂的 Makefile 文件，并且不需要仅为确保所有依赖关系被正确地更新而重新编译一切。

当至少一个“usage”仅在 *implementation*（实施）中时，在单元之间来制作一个循环（*circular*）依赖关系是被允许的。所以，在 *interface*（接口）中对于单元 A 来使用单元 B，然后在 *implementation*（实施）中单元 B 来使用单元 A 是被允许的。

3.2. 使用单元名称限定(Qualifying)标识符

不同的单元可能定义相同的标识符。为保持代码简单地读和查找，你应该总是避免它，但是它不是永远可能的。在这些情况下，在 *uses* 分句中的最后一个单元“获胜”，这意味着它引进的标识符隐藏由早期单元引进的相同的标识符。

你可以总是准确地定义一个已给标识符的单元，通过使用它，像 `MyUnit.MyIdentifier`。当你想从被其他单元隐藏的 `MyUnit` 使用标识符时，这是常用的解决方案。当然，你也可以重排在你 *uses* 分句上单元的次序，然而这可能影响其他的声明，而不是你尝试修复。

```

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

// Both Graphics and GoogleMapsEngine units define TColor type.
uses Graphics, GoogleMapsEngine;

var
  { This doesn't work like we want, as TColor ends up
    being defined by GoogleMapsEngine. }
  // Color: TColor;
  { This works Ok. }
  Color: Graphics.TColor;
begin
  Color := clYellow;
  Writeln(Red(Color), ' ', Green(Color), ' ', Blue(Color));
end.

```

在单元的中情况下，记住，它们有两个 **uses** 分句：一个在 **interface**（接口）中，另一个在 **implementation**（实施）中。规则：*后面的单元总是隐藏来自先前被应用这里的单元*，这意味着：*使用在 **implementation**（实施）分句中的单元可以隐藏来自使用在 **interface**（接口）分句中的单元的标识符*。然而，记住：当读 **interface** 分句时，仅使用在 **interface**（接口）事件 **matter** 中的单元，这可能创建一个混乱的情况，两个看起来相等的声明(declarations)经由编译器考虑成不同的：

```

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

// INCORRECT example

interface

uses Graphics;

procedure ShowColor(const Color: TColor);

implementation

uses GoogleMapsEngine;

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

end.

```

单元 **Graphics**(来自 **Lazarus LCL**)定义 **TColor** 类型。但是编译器将不能编译上面的单元，要求 (claiming)你不能实施(implement)一个匹配 **interface**(接口)声明(declarations)的 **procedure**(过

程)ShowColor。问题是单元 GoogleMapsEngine 已经定义一个 TColor 类型。并且它仅被使用在 implementation 分句中，因此它跟随(shadows)TColor 仅定义在 implementation(实施)中。上面单元的等效版本，错误是明显的，看起来像这个：

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

// INCORRECT example.
// This is what the compiler "sees" when trying to compile previous example

interface

uses Graphics;

procedure ShowColor(const Color: Graphics.TColor);

implementation

uses GoogleMapsEngine;

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

end.
```

在这种情况下，解决方案是无价值的(trivial)，只是更改 implementation(实施)到明确地使用来自 Graphics 单元的 TColor。当 UnitUsingColors 将要定义更多东西时，你也可以修复它，经过移动 GoogleMapsEngine 的使用，到 interface 分句并且在 Graphics 之前，然而在真实的世界的情况下可能导致其他因果关系(consequences)。

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

interface

uses Graphics;

procedure ShowColor(const Color: TColor);

implementation

uses GoogleMapsEngine;

procedure ShowColor(const Color: Graphics.TColor);
begin
    // Writeln(ColorToString(Color));
```

```
end;
```

```
end.
```

3.3. 从另一个单元接触(*expose*)一个单元标识符

有时，你想从一个单元获得一个标识符，并且在一个新的单元中 *接触(expose)* 它。最终的结果应该是：使用新的单元将使标识符在空间命名(namespace)中可用。

有时，与先前单元版本保持向后兼容是必要的。有时，这种方法"(隐藏)hide"一个内部单元是美好的。

可以在你的新的单元中重新定义标识符完成。

```
{ $mode objfpc } { $H+ } { $J- }  
unit MyUnit;  
  
interface  
  
uses Graphics;  
  
type  
  { Expose TColor from Graphics unit as TMyColor. }  
  TMyColor = TColor;  
  
  { Alternatively, expose it under the same name.  
    Qualify with unit name in this case, otherwise  
    we would refer to ourselves with "TColor = TColor" definition. }  
  TColor = Graphics.TColor;  
  
const  
  { This works with constants too. }  
  clYellow = Graphics.clYellow;  
  clBlue = Graphics.clBlue;  
  
implementation  
  
end.
```

注意：这个技巧不能与全局 procedures (过程)，函数和变量一样容易地被完成。使用 procedures(过程)和函数，你可以接触(*expose*)一个 *常量指针* 到一个在另外一个单元中的 procedure (过程) (查看 [Callbacks\(回调\)](#) (亦称 [events\(事件\)](#)，亦称 [pointers to functions\(指针到函数\)](#)，亦称 [procedural variables\(过程变量\)](#)))，但是这看起来非常令人不快(dirty)。

通常的解决方案是创建一个无价值的(trivial)"wrapper(包装器)"函数，它在下面简单地调用来自内部单元的函数，向各处(around)传递参数和返回值。

为使这与全局变量一起工作，可以使用全局(单元-级别)属性(properties)，查看 [属性\(Properties\)](#)。

4. 类

4.1. 基础

我们有类。在基础层，一个类只是一个容器，对于

- 字段 (*fields*) (它是一个对 “一个在类内部中的变量” 的设想 (fancy) 名称),
- 方法 (它是一个对 “一个在类内部中 *procedure* (过程) 或函数” 的设想 (fancy) 名称),
- 和 属性 (它是一个对一些看起来像一个字段 (*fields*), 但事实上是一对方法来 *get* (获取) 和 *set* (设置) 一些东西; 更多在 [属性](#))。
- 事实上, 这里有更多的可能, 描述在 [更多原材料 \(stuff\) 在类和嵌套类的内部](#).

```
type
  TMyClass = class
    MyInt: Integer;
    procedure MyMethod;
  end;

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

4.2. 继承 (Inheritance) , is, as

我们有继承 (inheritance) 和虚拟方法 (virtual methods) 。

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

uses SysUtils;

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

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

procedure TMyClass.MyVirtualMethod;
begin
  Writeln('TMyClass 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.

```

默认情况下，方法（methods）不是虚拟的（virtual），使用 **virtual** 声明它们来制作它们。重写（Overrides）必需用 **override** 标记，否则，你将得到一个警告。为不使用重写（overriding）（通常，你不想做这个，除非你知道（now）你在做什么）隐藏一个方法，使用 **reintroduce**（重新引入）。

为在运行时测试一个实例的类，使用 **is** 操作符。为将类型转换（**typecast**）实例到一个指定的类，使用 **as** 操作符。

```

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

uses SysUtils;

type
  TMyClass = class
    procedure MyMethod;
  end;

  TMyClassDescendant = class(TMyClass)
    procedure MyMethodInDescendant;
  end;

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

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

var

```

```

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

    { Descendant has all functionality expected of
      the TMyClass, so this assignment is OK }
    C := Descendant;
    C.MyMethod;

    { this cannot work, since TMyClass doesn't define this method }
    //C.MyMethodInDescendant;
    if C is TMyClassDescendant then
      (C as TMyClassDescendant).MyMethodInDescendant;

  finally FreeAndNil(Descendant) end;
end.

```

代替转换（casting）使用 `X as TMyClass`, 你也可以使用 *unchecked* 类型转换(typecast) `TMyClass(X)`。这是更快的, 但是, 如果 `X` 事实上不是一个 `TMyClass` 派生物, 导致一个未定义的行为。所以, 不要使用 `TMyClass(X)` 类型转换（typecast）, 或仅在一个明显的代码中使用它是正确的, 例如, 右侧在使用 `is` 测试后:

```

if A is TMyClass then
  (A as TMyClass).CallSomeMethodOfMyClass;
// below is marginally faster
if A is TMyClass then
  TMyClass(A).CallSomeMethodOfMyClass;

```

4.3. 属性

属性是一个非常好的“*syntax sugar*（语法糖）”, 对于

1. 使一些东西看起来像一个字段（`field`）(可以读和设置), 但是, 在下层通过调用一个 *getter*（获取）和 *setter*（设置）实现。典型的用法是, 每次一些值更改, 表现一些次要的效果（*side-effect*）(例如, 重绘制屏幕)。
2. 使一些东西看起来像一个字段（`field`）, 但是只读。在效果中, 它像一个常量或一个无参数的函数。

```

type
  TWebPage = class
  private
    FURL: string;
    FColor: TColor;
    function SetColor(const Value: TColor);

```

```

public
{ No way to set it directly.
  Call the Load method, like Load('http://www.freepascal.org/'),
  to load a page and set this property. }
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;
    // for example, cause some update each time value changes
    Repaint;
    // as another example, make sure that some underlying instance,
    // like a "RenderingComponent" (whatever that is),
    // has a synchronized value of Color.
    RenderingComponent.Color := Value;
  end;
end;

```

注意，代替指定一个方法，你也可以指定一个字段（`field`）（一般的是一个 `private`（私有的）字段（`field`））来直接获得或设置。在上面的示例中，`Color` 属性使用一个 *setter* 方法 `SetColor`。但是，对于获得值，`Color` 属性直接引用到 `private`（私有的）字段（`field`）`FColor`。直接引用到一个字段（`field`）快于实施（*implementing*）平常的 *getter* 或 *setter* 方法（对于你是较快的，在执行时是最快的）。

当声明一个你指定的属性时：

1. 不管它能读，并且如何读（通过直接读一个字段（`field`），或通过使用一个“*getter*”方法）。
2. 并且，在一个类似的方法中，不管它能设置，并且如何设置（通过直接写到一个指派的（*designated*）字段（`field`），或通过调用一个“*setter*”方法）。

编译器核查指示的字段（`field`）和方法的类型和参数与属性类型是否相配。例如，来读一个 `Integer`（整形）属性，你不得不要么提供一个 `Integer`（整形）字段（`field`），要么一个缺少参数的将返回一个 `Integer`（整形）的方法。

技术上，对于编译器，“*getter*”和“*setter*”方法仅是普通的方法，并且它们能够独立地做任何事（包括次要的效果或随机选择）。但是，设计属性来表现或多或少像字段（`fields`）是一个好的习惯来：

- *getter* 函数应该没有可视的次要的效果（例如，它不应该从文件/键盘读一些输入）。它应该是确定性的（非随机选择，甚至没有伪随机选择：）。如果在中间没有更改，多次读一

个属性应该是有效的，并且返回相同的值。

注意，对于 *getter* 来有一些 *非可视*次要的效果是可以的，例如，来缓存一些计算的一个值(已知对指定的实例来产生相同的结果)，下一次来快速返回它。事实上，这是一个 "getter" 函数的一个绝妙的可能性。

- *setter* 函数应该总是设置请求的值，这样的事物调用 *getter* 它返回的产生结果。在 "setter" (如果你必需，提引(raise)一个异常)中不要默默地拒绝无效值。不要转换或缩放 (scale) 需要的值。这个主意是在 `yClass.MyProperty := 123;` 之后的，程序员可以要求 `MyClass.MyProperty = 123`。
- *只读属性* 总是用于使一些来自外面的字段 (field) 只读。再一次，好的习惯是使它表现像一个习惯，至少用于这个对象实例的常量带有这个状态。属性的值不应该未料到的更改。如果使用它有一个次要的效果或返回任意的一些东西，使它是一个函数，而不是一个属性。
- 一个属性的 " (备份) *backing* " 字段 (field) 也总是 *private* (私有的)，尽管一个属性的想法是来封装 (encapsulate) 全部有权使用它的外部。
- 使 *仅设置属性* 是技术上可行的来，但是我还没有看到这样的一个好示例:)

注意	属性可以在一个单元层被类的外部定义。它们提供 (serve) 一个相似的 (analogous) 目的，然后，看起来像一个全局变量，但是通过一个 <i>getter</i> 和 <i>setter</i> 程序 (routines) 支持 (backed)。
----	---

4.4. 异常

我们有异常。它们可以被 `try ... except ... end` 分句捕获，并且我们有 `finally` 分句，像 `try ... finally ... end`。

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

```
program MyProgram;
```

```
uses SysUtils;
```

```
type
```

```
  TMyClass = class
```

```
    procedure MyMethod;
```

```
  end;
```

```
procedure TMyClass.MyMethod;
```

```
begin
```

```
  if Random > 0.5 then
```

```
    raise Exception.Create('Raising an exception!');
```

```
end;
```

```
var
```

```
  C: TMyClass;
```

```
begin
```

```
  C := TMyClass.Create;
```

```
try
  C.MyMethod;
finally FreeAndNil(C) end;
end.
```

注意，哪怕你使用 `Exit` (从函数 / procedure (过程) / 方法) 或 `Break` 或 `Continue` (从循环体) 退出语句块，`finally` 分句将被执行。

4.5. Visibility specifiers (可视化说明符/分类符)

像在大多数面向对象语言中，我们有 visibility specifiers (可视化说明符/分类符) 来隐藏字段 (fields) / 方法 / 属性。

基础的可视化等级是：

`public`

每个人可以访问它，包含在其它单元中的代码。

`private`

仅在这个类中可访问。

`protected`

仅在这个类和其派生类中可访问。

`private` (私有的) 和 `protected` (受保护的) 的上述可视性的解释不是严格正确的。在相同单元中的代码可以克服它们的限制，自由地访问私有的 (`private`) 和受保护的 (`protected`) 的原料 (stuff)。有时，这是一个极好的特征，允许你来实现 (implement) 紧密连接的 (tightly-connected) 类。使用绝对私有 (`private`) 或绝对受保护的 (`protected`) 来更紧密地保护你的类。看 [私有 \(private\) 和绝对私有 \(private\)](#)。

默认情况下，如果你不指定可视性，那么声明的原料 (stuff) 的可视性是公共的 (`public`)。例外，类带有 `{ $M+ }` 被编译，或类的衍生物带有 `{ $M+ }` 被编译，这包含 `TPersistent` 的所有的衍生物，这也包含 `TComponent` 的所有的衍生物 (尽管 `TComponent` 起源自 `TPersistent`)。对于它们，默认可视性分类符 (specifier) 是 `published` (公共的)，这像 `public` (公共的)，但另外，流系统知道处理这个。

不是每一个字段 (field) 和属性类型在 `published` 部分中是被允许的 (不是每一个类型可以被流化 (streamed)，仅类可以从简单的字段 (fields) 中被流化 (streamed))。如果你不关心流，但是想一些事对所有用户可用，仅仅使用 `public` (公共的)。

4.6. 默认 原型 (ancestor)

如果你不声明原型 (ancestor) 类型，每一个类 继承自 `TObject`。

4.7. Self

这个特殊的关键字 `Self` 可以被使用在类实施 (implementation) 来 [明确地适用于 /// 显示引用 ///](#) 你自己的实例。它与来自 C++，Java 和相似语言的 `this` 相当。

4.8. 调用继承的方法(inherited method)

在一个方法实施(implementation)中，如果你调用另一个方法，那么默认你调用你自己类的方法。在下面的示例代码中，TMyClass2.MyOtherMethod 调用 MyMethod,这以调用 TMyClass2.MyMethod 结束。

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

如果方法不是定义在一个所给的类，那么它调用一个原型（ancestor）类的一个方法。事实上，当你在 TMyClass2 的一个实例上调用 MyMethod 时，到那时

- 编译器寻找 TMyClass2.MyMethod。

- 如果没有找到，它寻找 TMyClass1.MyMethod。
- 如果没有找到，它寻找 TObject.MyMethod。
- 如果没有找到，那么，编译失败。

在上面的示例中，你可以测试它，通过注释掉 TMyClass2.MyMethod 定义。事实上，TMyClass1.MyMethod 将经由 TMyClass2.MyOtherMethod 调用。

有时，你不希望调用你拥有的类的方法。你希望调用一个原型（或原型的原型，等等）的方法。为了做到这个，在调用 MyMethod 前添加关键字 inherited 来调用 MyMethod，像这样：

inherited MyMethod;

这种方法，你强迫编译器来从一个原型类中开始搜索。在我们的示例中，它意味着编译器在 TMyClass1.MyMethod 中，其次在 TObject.MyMethod 中搜索 MyMethod，然后放弃。它甚至不考虑使用 TMyClass2.MyMethod 的实施(implementation)

提示	前进，使用继承的 MyMethod 更改上面的 TMyClass2.MyMethod 的实施(implementation)，并在输出中看不同。
----	--

inherited 调用经常被用于调用相同名称的原型(ancestor)方法。这种方法派生物(descendants)可以增强原型(ancestors) (保持原型(ancestor)函数性，而不是替换原型(ancestor)函数性)。像在下方的示例中。

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

```

尽管使用 `inherited` 来调用一个带有相同名称、带有相同参数的方法是非常经常的情况，对它有一个指定的快捷方式：你可以只写 `inherited;` (`inherited` 关键字直接跟随一个分号，而不是一个方法名称)。这意味着“调用一个继承的带有相同名称的方法，作为当前的方法传递它相同的参数”。

提示	在上面的示例中，所有的 <code>inherited ...;</code> 调用可以被一个简单的 <code>inherited;</code> 代替。
----	--

注释 1: `inherited;` 对于调用带有 *相同*变量传入的原型(ancestor)的方法真的只是一个快捷方式。如果你已经修改你自己的参数(这是可能的，如果参数不是常量)，那么原型(ancestor)的方法可以从你的衍生物中接收不同的输入值。考虑这个：

```

procedure TMyClass2.MyMethod(A: Integer);
begin
    Writeln('TMyClass2.MyMethod beginning ', A);
    A := 456;
    { This calls TMyClass1.MyMethod with A = 456,
      regardless of the A value passed to this method (TMyClass2.MyMethod). }
    inherited;
    Writeln('TMyClass2.MyMethod ending ', A);
end;

```

注释 2: 当很多类(沿着“继承链”)定义它时，你通常想使 `MyMethod` 虚拟的(*virtual*)。更多的虚拟(*virtual*)方法在下面部分中。但是不管是否该方法是虚拟的(*virtual*)或不是，`inherited` 关键字工作。`inherited` 总是意味着，编译器对在一个原型(ancestor)中的方法开始搜索，并且它对虚拟的(*virtual*)和非虚拟的(*not virtual*)方法都讲得通。

4.9. 虚拟 (Virtual) 方法，override (重写) 和 reintroduce (再引入)

默认情况下，方法不是虚拟的 (*virtual*)。这类似于 C++，并不像 Java。

当一个方法不是虚拟的 (*virtual*) 时，编译器基于当前声明的(*declared*)类的类型决定哪个方

法来调用，而不是基于事实上创建的类的类型。不同点看起来难以捉摸，但是当你的变量被声明到有一个类时是重要的，像 TFruit，但是它实际上可以是一个衍生物(descendant)类，像 TApple。

面向对象编程的思想(idea)是衍生物(descendant)类总是和原型(ancestor)一样好,所以当原型(ancestor)被要求时，编译器总是允许使用一个衍生物(descendant)类。当你的方法不是虚拟的(virtual)，这可能有非期望的结果(consequences)。考虑下面的示例：

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

type
  TFruit = class
    procedure Eat;
  end;

  TApple = class(TFruit)
    procedure Eat;
  end;

procedure TFruit.Eat;
begin
  Writeln('Eating a fruit');
end;

procedure TApple.Eat;
begin
  Writeln('Eating an apple');
end;

procedure DoSomethingWithAFruit(const Fruit: TFruit);
begin
  Writeln('We have a fruit with class ', Fruit.ClassName);
  Writeln('We eat it:');
  Fruit.Eat;
end;

var
  Apple: TApple; // Note: you could as well declare "Apple: TFruit" here
begin
  Apple := TApple.Create;
  try
    DoSomethingWithAFruit(Apple);
  finally FreeAndNil(Apple) end;
end.
```

这个示例将打印

We have a fruit with class TApple

We eat it:
Eating a fruit

事实上，调用称为 `TFruit.Eat` 实施(implementation)的 `Fruit.Eat`，并毫不调用 `TApple.Eat` 实施(implementation)。

如果你回想起编译器如何工作，这是合乎自然规则的：当你写 `Fruit.Eat`，`Fruit` 变量被声明来容纳一个类 `TFruit`。因此编译器在 `TFruit` 类中搜索称为 `Eat` 的方法。如果 `TFruit` 类不能包含这样的方法，编译器将在一个原型(ancestor) (`TObject` 在这种情况下)中搜索，编译器将在一个中搜索。但是编译器不能在衍生物(descendants) (像 `TApple`)中搜索，因为它不知道 `Fruit` 的实际的(actual)类是 `TApple`，`TFruit` 中的哪一个，或者其它 `TFruit` 衍生物(descendants) (像一个 `TOrange`，不显示在上面的示例中)。

换句话说，将被调用的方法是在编译时被决定的。

使用虚拟方法(virtual methods)更改这个行为(behavior)。如果 `Eat` 方法将是虚拟的(virtual) (它的一个示例是在下面显示)，那么实际将被调用的实施(implementation) 在运行时被决定。如果 `Fruit` 变量将容纳一个 `TApple` (哪怕它被声明为 `TFruit`)类的实例，那么 `Eat` 方法将首先在 `TApple` 类中搜索。

在 Object Pascal 中，来定义一个方法作为 *virtual*(虚拟的)，你需要

- 用 `virtual` 关键字标记它的第一个定义(definition) (在最顶级的原型(ancestor))。
- 用 `override` 关键字比较所有的其它定义(definitions) (在衍生物(descendants))。所有重写(overridden)版本必需有完全(exactly)相同的参数(并返回相同的类型，在函数的情况下)。

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

type
  TFruit = class
    procedure Eat; virtual;
  end;

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

procedure TFruit.Eat;
begin
  Writeln('Eating a fruit');
end;

procedure TApple.Eat;
begin
  Writeln('Eating an apple');
end;

procedure DoSomethingWithAFruit(const Fruit: TFruit);
begin
  Writeln('We have a fruit with class ', Fruit.ClassName);
```

```

    Writeln('We eat it:');
    Fruit.Eat;
end;

var
    Apple: TApple; // Note: you could as well declare "Apple: TFruit" here
begin
    Apple := TApple.Create;
    try
        DoSomethingWithAFruit(Apple);
    finally FreeAndNil(Apple) end;
end.

```

这个示例将打印

We have a fruit with class TApple

We eat it:

Eating an apple

本质上，虚拟方法通过有所谓的*虚拟方法表* (*virtual method table*)与每个类关联而工作。这个表是这个类的指针到虚拟方法的实施(implementations)的一个列表。当调用 `Eat` 方法时，编译器查看与 `Fruit` 的实际的类相关联的一个虚拟方法表，并使用一个指针到存储在这里的 `Eat` 实施(implementations)。

如果你不使用 `override`(重写)关键字，编译器将警告你，你正在*隐藏(遮蔽)*一个带有非虚拟定义的原型(ancestor)的虚拟方法。如果你确定这是你想要的，你可以添加一个 `reintroduce`(再引入)关键字。但是在大多数情况下，你反而想保持该方法虚拟，并添加 `override`(重写)关键字，以此方法确保它总是正确的引用(invoked)。

5. 释放类

5.1. 记住释放类实例

类实例不得不手动释放，否则你将得到内存泄露。我建议使用 FPC -gl -gh 选项来侦查 (detect) 内存泄露(查看 https://castle-engine.io/manual_optimization.php#section_memory).

注意，这不涉及提引 (raised) 异常。尽管当提引 (raising) 一个异常时，你确实创建一个类（并且它是一个完美的正常的类，并且你也可以创建你拥有的为这个目的类）。但是这个类实例是自动释放的。

5.2. 如何释放

为释放类实例，最好在你的类实例上调用 `FreeAndNil(A)`。它检查是否 `A` 是 `nil`，如果不是—调用它的析构函数 (destructor)，并且设置 `A` 到 `nil`。所以在一个行中调用它很多次不是一个错误。

它是或多或少的一个快捷方式 (shortcut)，对于

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

事实上，这是一个过度简化的，在一个适合的引用 (reference) 上，因为 `FreeAndNil` 做了一个有用的骗局(trick)，并在调用析构函数 (destructor) 前设置变量 `A` 为 `nil`。这帮助阻止某一个类的错误 (bugs) —这个主意是，"外部的" 代码应该永远不能访问类的一个半破坏的 (half-destructed) 实例。

你将经常看到人们使用 `A.Free` 方法。这是像做

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

这释放 `A`，除非它是 `nil`。

注意，在正常情况下，在一个可能是 `nil` 的实例上，你应该永远不调用一个方法。所以，如果 `A` 可能是 `nil`，乍看之下，调用 `A.Free` 可能看起来令人怀疑。然而，`Free` 方法是这个规则的一个例外。在实施 (implementation) 中它做的有些卑鄙 (dirty) — 也就是，检查是否 `Self <> nil`。这个卑鄙的骗局 (dirty trick) 仅工作在非虚拟 (non-virtual) 方法中(这不调用任何的虚拟方法和不访问任何的字段 (fields))。

我建议总是使用 `FreeAndNil(A)`，没有例外的情况，并永远不直接地调用 `Free` 方法或 `Destroy` 析构函数 (destructor)。 *Castle Game Engine* 像这样做的。它帮助保持一个精确的断言 (assertion)，所有的引用 (references) 不是 `nil`，就是指向有效的实例。.

5.3. 手动和自动释放

在很多情况下，需要释放实例不是多少问题。你只是 (just) 写一个析构函数 (destructor)，这匹配一个构造函数 (constructor)，并解除重新分配 (deallocates) 被分配在构造函数 (constructor) 中的一切 (或，更彻底，在类的整个使用期限)。注意，仅释放每个东西一次。通常，设置释放的引用 (reference) 为 `nil` 是一个好主意，通常，通过调用 `FreeAndNil(A)` 来

做它是最安逸的.

所以,像这个:

```
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;
```

为避免明确地(explicitly)释放实例,也可以使用"ownership"的 `TComponent` 特征.一个被拥有的对象将自动地通过拥有者释放.该机制是智能的,并且它将永远不会释放一个应该释放的实例(因此,如果你早期手动释放拥有的对象,事情也将正确地工作).我们可以更改前一个示例到这个:

```
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;

```

注意：我们需要在这里重写(override)一个虚拟 TComponent 构造函数 (constructor)。所以我们不更改构造函数 (constructor) 参数。(事实上，你可以一用 reintroduce(再引入)声明一个新的构造函数 (constructor)。但是注意，由于一些功能,例如.streaming(流),将仍然使用虚拟构造函数 (constructor)，所以，在任何情况下确保它正确地工作.)

自动释放的另一种机制是 list-classes(列表-类)的 OwnsObjects 功能/(默认已经是 true!),像 TFPGObjectList 或 TObjectList.所以我们可以写：

```

uses SysUtils, Classes, FGL;

type
  TGun = class
  end;

  TGunList = specialize TFPGObjectList<TGun>;

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

constructor TPlayer.Create;
begin
  inherited;
  // Actually, the parameter true (OwnsObjects) is already the default
  Guns := TGunList.Create(true);
  Gun1 := TGun.Create(Self);
  Guns.Add(Gun1);
  Gun2 := TGun.Create(Self);
  Guns.Add(Gun2);
end;

destructor TPlayer.Destroy;
begin
  { We have to take care to free the list.
    It will automatically free it's contents. }
  FreeAndNil(Guns);

  { No need to free the Gun1, Gun2 anymore. It's a nice habit to set to "nil"
    their references now, as we know they are freed. In this simple class,
    with so simple destructor, it's obvious that they cannot be accessed
    anymore -- but doing this pays off in case of larger and more complicated

```

destructors.

Alternatively, we could avoid declaring Gun1 and Gun2,
and instead use Guns[0] and Guns[1] in own code.

Or create a method like Gun1 that returns Guns[0]. }

Gun1 := nil;

Gun2 := nil;

inherited;

end;

注意, 列表类(list classes) "ownership"机制是简单的,并且, 如果你使用一些其他的方法释放实例, 你将获得一个错误,尽管它也包含在一个列表中.使用 `Extract` 方法来从一个列表移除一些东西而不释放它,因此,你自己有责任来释放它.

在 **Castle Game Engine** 中: 当作为另一个 `TX3DNode` 的子类(children)插入时,`TX3DNode` 的派生类(descendants)有自动内存管理. 根 `X3D` 节点, `TX3DRootNode`, 通常是依次由 `TCastleSceneCore` 拥有.其它的一些事物也有一个简单的所有权机制 — 查看称为 `OwnsXxx` 的参数和属性.

5.4. 虚拟析构函数(destructor)被称为 Destroy(销毁)

如你在上面示例所见, 当类被销毁时, 它的析构函数 (destructor) 被称为 `Destroy`(销毁)被调用.

理论上,你可以有多个析构函数(destructors), 但是,在实践中,它几乎从来不是一个好主意.仅有一个称为 `Destroy` 的析构函数(destructors)更容易, `Destroy` 依次通过 `Free` 方法被调用, `Free` 方法依次通过 `FreeAndNil` 过程被调用.

`Destroy` 析构函数(destructors)在 `TObject` 中被定义为一个虚拟方法, 所以你应该总是在你的类 (尽管所有的类衍生(descend)自 `TObject`)中用 `override` 关键字标记它. 这使 `Free` 方法正确地工作. 回想起,虚拟方法方法如何工作,从 [虚拟\(Virtual\)方法, override\(重写\) 和 reintroduce\(再引入\)](#).

注意	<p>这个信息是关于析构函数(destructors), 实际上,与构造函数(constructors)不符.</p> <p>它的正常的,一个类有多个构造函数(constructors). 通常它们被称为 <code>Create</code>, 只是有不同的参数, 但是,它也是很好的来为构造函数(constructors)创造其它的名称.</p> <p>而且, <code>Create</code> 构造函数(constructors)在 <code>TObject</code> 中不是虚拟的(virtual), 所以,你不要在派生物中用 <code>override</code> 标记它.</p> <p>当定义构造函数(constructors)时,这一切给你一点额外的灵活性.使它们虚拟常常不是不要的, 所以,默认情况下,你不要强制做它.</p> <p>注意,无论如何,这些对 <code>TComponent</code> descendants 派生物更改. <code>TComponent</code> 定义一个虚拟构造函数(constructors) <code>Create(AOwner: TComponent)</code>. 为了流(streaming)系统来工作,它需要一个虚拟构造函数(constructors). 当定义 <code>TComponent</code> 的派生物时, 你应该重写(override)这个构造函数(constructors) (并且用 <code>override</code> 关键字标记它), 并在它的内部执行所有你的初始化.它仍然是很好的来定义附加的(additional) 构造函数(constructors), 但是它们仅应该充当"帮手(helpers)". 当使用 <code>Create(AOwner: TComponent)</code> 构造函数(constructors)创建时,这个实例也应该工作, 否则,当流(streaming)时,它将不能被正确的构造(constructed). 流(streaming)被使用. 例如,当在一个 <code>Lazarus</code> 窗体上保存和加载这个组件时.</p>
----	---

5.5. 释放 notification(通知)

如果你复制一个引用(reference)到实例, 这样, 你有两个引用(reference)到相同的存储器, 然后它们中的一个被释放—另一个变成一个“悬挂指针”. 它不能被访问, 因为它指向到的一个不再被分配的存储器. 访问它可能导致在一个运行时中的错误, 或者无用的数据被返回(returned) (因为存储器可以被你的程序中的其它原料(stuff)重新利用).

在这里使用 `FreeAndNil` 来释放实例没有帮助. `FreeAndNil` 设置为 `nil` 仅引用(reference)它获得—没有方法对它来设置所有其他的引用(reference). 考虑这代码:

```
var
  O1, O2: TObject;
begin
  O1 := TObject.Create;
  O2 := O1;
  FreeAndNil(O1);

  // what happens if we access O1 or O2 here?
end;
```

1. 在这个语句块的结尾, `O1` 是 `nil`。如果一些代码不得不访问它, 它可以可靠地使用 `if O1 <> nil ...` 来避免在一个释放的实例上调用方法, 像 `if O1 <> nil then Writeln(O1.ClassName);`

尝试访问一个 `nil` 实例的一个字段导致一个可预见的在运行时(runtime)处的异常. 因此, 虽然一些代码将不核对 `O1 <> nil`, 并将盲目地访问 `O1` 字段, 你将在运行时(runtime)处获得一个明确的异常.

相同的, 去调用一个虚拟(virtual)方法, 或调用一个非虚拟(non-virtual)方法来访问一个 `nil` 实例的字段.

2. 对于 `O2`, 事情变的较少地可预见的. 它不是 `nil`, 但是它是无效的. 尝试来访问一个非无(non-nil)无效实例的一个字段导致一个不可预见的行为—可能一个访问违例异常, 可能一个垃圾数据被返回(returned).

有属于它各种各样的的解决方案:

- 一个解决方案是来, 完全地, 仔细的和阅读文档. 不要假设关于引用(reference)的生命时间(lifetime)的任何事, 如果它通过其它代码被创建. 如果一个类 `TCar` 有一个字段指向 `TWheel` 的一些实例, 它是一个惯例(convention), 引用(reference)到 `wheel` 是有效的, 尽管引用(reference)到 `car` exists 继续存在, 并且 `car` 将在它的析构函数(destructor)内部释放它的 `wheels`. 但是这只是一个惯例(convention), 文档应该涉及, 如果这里有更复杂的一些事进行.
- 在上面的示例中, 在恰当地释放 `O1` 实例后, 你可以简单地明确地设置 `O2` 变量为 `nil`. 在这个简单的情况下是不重要的.
- 最 future-proof 解决方案是来使用 `TComponent` 类“释放通知(notification)”机制. 一个组件可以被通知(notified), 当另外的组件被释放, 并因此设置它的引用(reference)为 `nil`. 于是你获得一些事, 像一个弱引用(weak reference). 它可以对付各种各样的用法事态, 例如你可以让来自类的外部的代码来设置你的引用(reference), 并且外部的代码也可以在任何时间释放实例.

这需要两类都派生自 `TComponent`. 一般而言使用它归结到调用 `FreeNotification`,

RemoveFreeNotification, 并重写 Notification.

这里是一个完整的示例,显示如何使用这个机制,与构造函数(constructor) /析构函数和一个 setter 属性一起.有时它可以做的更简单,但是这是成熟的版本,换句话说,总是正确的:)

```
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;
end;

destructor TContainer.Destroy;
begin
  { set to nil by SetSomeSpecialControl, to detach free notification }
  SomeSpecialControl := nil;
end;
```

```
inherited;  
end;
```

6. 运行时库

6.1. 使用 streams（流）输入/输出

现代的程序应该使用 `TStream` 类，并且它是很多完成输入/输出的衍生物。它有很多有用的派生物，像 `TFileStream`, `TMemoryStream`, `TStringStream`.

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

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

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

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

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

在 Castle Game Engine 中： 你应该使用 `Download` 方法来创建一个流(stream), 操作资源(资源包括下载自 URLs 和 Android 资产(assets)的文件，数据)。此外，为打开在你游戏数据(一般在 `data` 子目录中)中的资源使用 `ApplicationData` 函数。

```
EnableNetwork := true;
S := Download('https://castle-engine.io/latest.zip');
S := Download('file:///home/michalis/my_binary_file.data');
S := Download(ApplicationData('gui/my_image.png'));
```

为读文本文件，我们建议使用 `TTextReader` 类。它提供一个面向行的(line-oriented)API，并且包裹(wrap)一个 `TStream` 在里面。`TTextReader` 构造函数可以获取一个准备好的 URL，或你可以传递到你自定义的 `TStream` 源文件那里。

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

6.2. 容器 (lists(列表), dictionaries(词典))使用泛型(generics)

Pascal 语言和运行时库提供各种各样灵活的容器(containers)。有许多的非-泛型类(像 `TList` 和 `TObjectList` 来自 `Contrns` 单元)，也有动态数组(array of `TMyType`)。但为了获得最大的灵活性

和类型-安全，对你需要的大多数，我建议使用泛型容器。

泛型容器给你大量有用的方法来 add(添加)，remove(移除)，iterate(迭代)，search(查找)，sort(排序)... 编译器也知道(和检查)，容器仅保存恰当类型的项目。

现在，在 FPC 中有 3 个库提供泛型容器：

- FGL 单元
- Generics.Collections 单元和近亲(friends)
- GVector 单元和近亲(friends) (一起在 fcl-stl 中)

我们建议使用 FGL 单元(如果你需要与稳定的 FPC 3.0.x 或甚至更老的 FPC 2.6.x 一起工作)，或 Generics.Collections 单元(仅自 FPC 3.1.1 可用，但是在其他方面兼容 Delphi)。它们都提供 lists (列表) 和 dictionaries (词典) 带有名义上组成标准库的其它部分(像来自 Contnrs 单元的非-泛型容器)。

在 Castle Game Engine 中： 我们包含一个 Generics.Collections 的本地复制本，即使对于 FPC 3.0.x.在引擎期间，我们使用 Generics.Collections，建议你也使用 Generics.Collections !

来自 Generics.Collections 单元最重要的类是：

TList (列表)

一个泛型类型的列表。

TObjectList (对象列表)

一个泛型对象实例(instances)的列表。它可以“拥有”子类(children)，这意味着它将自动释放(free)它们。

TDictionary (词典)

一个泛型词典。

TObjectDictionary (对象词典)

一个泛型词典，它可以“拥有”字(keys)和/或值(values)。

这里是您如何使用一个简单的泛型 TObjectList:

```
{ $mode objfpc } { $H+ } { $J- }  
uses SysUtils, Generics.Collections;  
  
type  
  TApple = class  
    Name: string;  
  end;  
  
  TAppleList = specialize 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.

```

注意，一些操作需要比较两个项目，像排序和查找(例如，经过 `Sort` 和 `IndexOf` 方法)。

`Generics.Collections` 容器用于这个 *comparer* (比较器)。默认 *comparer* (比较器) 对所有类型是适当的，甚至对于记录 (records) (在这种情况下，它比较存储器内容。至少对于搜索使用 `IndexOf`，它是一个合理的默认)。

当排序列表时，你可以通过一个自定义 *comparer* (比较器) 作为一个参数。*comparer* (比较器) 是一个类，实施 (implementing) `IComparer` 接口 (interface)。在实践中，你经常定义适当的 *callback* (回调)，并使用 `TComparer<T>.Construct` 方法来包裹 (wrap) 这个 *callback* (回调) 进入到一个 `IComparer` 实例。做这个的一个示例在下面：

```

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

type
  TApple = class
    Name: string;
  end;

  TAppleList = specialize TObjectList<TApple>;

function CompareApples(constref Left, Right: TApple): Integer;
begin
  Result := AnsiCompareStr(Left.Name, Right.Name);
end;

type
  TAppleComparer = specialize 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(@CompareApples));

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

```

`TDictionary` 类 implements(实施)一个 **dictionary** (词典), 也被认为一个 **map** (地图) (**key** (键) → **value** (值)), 也被认为一个相关联的 (**associative**) 数组。它的 API 是一些类似于 C# `TDictionary` 类。它有有用的迭代器 (iterators), 对于 **keys** (键), **values** (值), 和成对的 **key** (键) → **value** (值)。

一个使用 **dictionary** (词典) 的示例代码:

```

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

type
  TApple = class
    Name: string;
  end;

  TAppleDictionary = specialize TDictionary<string, TApple>;

var
  Apples: TAppleDictionary;
  A, FoundA: TApple;
  ApplePair: TAppleDictionary.TDictionaryPair;
  AppleKey: string;
begin
  Apples := TAppleDictionary.Create;
  try
    A := TApple.Create;

```

```

A.Name := 'my apple';
Apples.AddOrSetValue('apple key 1', A);

if Apples.TryGetValue('apple key 1', FoundA) then
    Writeln('Found apple under key "apple key 1" with name: ' +
        FoundA.Name);

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

{ Line below works too, but it can only be used to set
  an *existing* dictionary key.
  Instead of this, usually use AddOrSetValue
  to set or add a new key, as necessary. }
// Apples['apple key 1'] := ... ;

Apples.Remove('apple key 1');

{ Note that the TDictionary doesn't own the items,
  you need to free them yourself.
  We could use TObjectDictionary to have automatic ownership
  mechanism. }
A.Free;
finally FreeAndNil(Apples) end;
end.

```

`TObjectDictionary` 可以附加地拥有 dictionary（词典）keys（键）和/或 values（值），这意味着，它们可以被自动的释放（freed）。如果它们是对象实例，注意仅拥有的 keys（键）和/或 values（值）。如果设置到 "owned" 一些其他类型，像一个 `Integer`（整型）（例如，如果你的 keys（键）是 `Integer`（整型），你包含（include）`doOwnsKeys`），当代码执行时，你将获得一个令人讨厌的崩溃。

一个使用 `TObjectDictionary` 的示例代码在下面。编译这个代码使用 *memory leak detection*（存储器泄漏检查），像 `fpc -gl -gh generics_object_dictionary.lpr`，来查看，当出现退出时，有关的事情被释放。

```

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

type
    TApple = class
        Name: string;
    end;

```

```

TAppleDictionary = specialize TObjectDictionary<string, TApple>;

var
  Apples: TAppleDictionary;
  A: TApple;
  ApplePair: TAppleDictionary.TDictionaryPair;
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.

```

如果你更喜欢使用 FGL 单元，而不是 Generics.Collections，这是来自 FGL 单元的最重要类的一个简短的概述：

TFPGList

一个泛型类型的列表。

TFPGObjectList

一个泛型对象实例的列表。它可以"拥有"children（孩子）。

TFPGMap

一个泛型 dictionary（词典） ‘

在 FGL 单元中，TFPGList 仅能被用于被定义等于运算(=)的定义。对于 TFPGMap，“大于”(>)和“小于”(<)运算必需被定义用于 key（键）类型。如果你想使用这些带有类型的列表，不能有内建的比较运算符(例如，带有纪录(records))，你不得不在 [Operator overloading（运算符重载）](#) 中显示，overload（重载）它们的运算符。

在 **Castle Game Engine** 中，我们含有一个单元 CastleGenericLists，它添加 TGenericStructList 和 TGenericStructMap 类。它们类似于 TFPGList 和 TFPGMap，但是它们不需要（require）一个比较运算符的定义，用于适当的类型（作为代替，它们比较存储器内容，对于纪录(records)或 method pointers（方法指针），它们经常是适当的）。CastleGenericLists 单元是独立的（deprecated），但是自从引擎版本 6.3 起，as 我们建议使用 Generics.Collections 代替。

如果你想知道更多关于 generics（泛型）,查看 [Generics（泛型）](#)。

6.3. Cloning（克隆）: TPersistent.Assign

复制类实例，通过一个简单的赋值运算复制 **reference**（引用）。

```
var
  X, Y: TMyObject;
begin
  X := TMyObject.Create;
  Y := X;
  // X and Y are now two pointers to the same data
  Y.MyField := 123; // this also changes X.MyField
  FreeAndNil(X);
end;
```

为复制类实例内容，标准的途径（approach）是从 TPersistent 获得你的类，并且 override（重写/覆盖）它的 Assign（赋值）方法。一旦它在 TMyObject 中适当地实施（implemented），你使用它，像这个：

```
var
  X, Y: TMyObject;
begin
  X := TMyObject.Create;
  Y := TMyObject.Create;
  Y.Assign(X);
  Y.MyField := 123; // this does not change X.MyField
  FreeAndNil(X);
  FreeAndNil(Y);
end;
```

为使它工作，你需要 implement(实施)Assign(赋值)方法来实际上地复制你想要的字段（fields）。你应该仔细地 implement（实施）Assign（赋值）方法，来从一个类中复制，这可能是当前类的一个派生类（descendant）。

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

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

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

```

procedure TMyClass.Assign(Source: TPersistent);
var
    SourceMyClass: TMyClass;
begin
    if Source is TMyClass then
    begin
        SourceMyClass := TMyClass(Source);
        MyInt := SourceMyClass.MyInt;
        // Xxx := SourceMyClass.Xxx; // add new fields here
    end else
    { Since TMyClass is a direct TPersistent descendant,
      it calls inherited ONLY when it cannot handle Source class.
      See comments below. }
    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; // add new fields here
    end;

    { Since TMyClassDescendant has an ancestor that already overrides
      Assign (in TMyClass.Assign), it calls inherited ALWAYS,
      to allow TMyClass.Assign to handle remaining fields.
      See comments below for a detailed reasoning. }
    inherited Assign(Source);
end;

var
    C1, C2: TMyClass;
    CD1, CD2: TMyClassDescendant;
begin
    // test TMyClass.Assign
    C1 := TMyClass.Create;
    C2 := TMyClass.Create;
    try
        C1.MyInt := 666;
        C2.Assign(C1);
    end;
end;

```

```

    Writeln('C2 state: ', C2.MyInt);
finally
    FreeAndNil(C1);
    FreeAndNil(C2);
end;

// test 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.

```

一些时候，它在源文件类中更适合在源文件类中来代替（alternatively）override（重写）AssignTo 方法，而不是在派生类（destination）类中 overriding（重写）Assign 方法。注意，当你调用在 overridden（重写）Assign（赋值）implementation（实施）中的 inherited（继承）时。这里有两种情况：

你的类是 TPersistent 类的直接派生类（descendant）。(或,它不是 TPersistent 的一个直接派生类（descendant），但是无 ancestor（原型/祖先）overridden（重写）Assign（赋值）方法。)

在这种情况下，你的类应该使用 inherited（继承）关键字(来调用 TPersistent.Assign)，*仅假如你不能在你的代码中处理（handle）assignment（赋值）。*

你的类从一些类派生（descends），已经 overridden（重写）Assign（赋值）方法。

在这种情况下，你的类应该总是使用 inherited（继承）关键字(来调用 ancestor（原型/祖先）Assign)。通常，在 overridden（重写）方法中调用 inherited（继承）通常是一个好主意。

为理解需要，当从 Assign（赋值）implementation（实施）中调用(或不调用) inherited（继承），如何关联（relates）到 AssignTo 方法，最好查看 TPersistent.Assign 和 TPersistent.AssignTo implementations（实施）：

```

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;

```

这不是 TPersistent 的**严密的 implementation**（实施），我简化它来隐藏关于异常信息如何注意 被构建的令人厌烦的细节。

结论，你可以从上面得到，是：

- 如果既不 Assign 也不 AssignTo 被 overridden（重写），那么调用它们将导致一个异常。
- 也注意，在 TPersistent implementation（实施）中无代码，它自动地复制类的所有 fields（字段）(或所有 published（公布的）fields（字段）)。这是为什么你需要你自己做它，通过在所有类中 overriding（重写）Assign（赋值）。你可以使用 RTTI(运行时类型信息)对这些，但是对于简单情况，你将很可能仅列表将被手动复制的字段（fields）。

当你有一个类，像 TApple，你的 TApple.Assign implementation（实施）通常处理复制 fields（字段），这被指定到 TApple 类(不是到 TApple ancestor（原型/祖先），像 TFruit)。所以，在复制 apple-相关的 fields（字段）前，TApple.Assign implementation（实施）通常在开始处（beginning）检查源文件是 TApple。然后，它调用 inherited（继承）来允许 TFruit 来处理（handle）fields（字段）的剩余部分（rest）。

假设，你 implemented（实施）TFruit.Assign 和 TApple.Assign 下列的（following）标准模式（pattern）(像在上面的实例中显示)，影响/效果（effect）是像这个：

- 如果你传递 TApple 实例到 TApple.Assign，它将工作并复制所有的 fields（字段）。
- 如果你传递 TOrange 实例到 TApple.Assign，它将工作并复制 common fields shared by both TOrange and TApple。换句话说，在 TFruit 处定义的 fields（字段）。
- 如果你传递 TWerewolf 实例到 TApple.Assign，它将 raise（提引）一个异常(因为 TApple.Assign 将调用 TFruit.Assign，它将调用 TPersistent.Assign，它 raises（提引）一个异常)。

注意	记住，当从 TPersistent 派生(descending)，默认 <i>visibility specifier</i> （可视化说明符/分类符）是 published（公布的），来允许 TPersistent 派生类（descendants）的 streaming（流）。不是所有的 field（字段）和属性类型被允许在 published（公布的）部分（section）。如果你获得相关到它错误，并且你不关心 streaming（流），仅更改 visibility（可视化）到 public（公共）。查看 Visibility specifiers（可视化说明符/分类符） 。
----	---

7. 各种各样的语言特征

7.1. 局部(嵌套)例行程序 (routines)

在大量的例行程序(routine)(函数,procedure(过程),方法)中,你可以定义一个帮助器(helper)例行程序(routine).

局部(嵌套)例行程序(routine)可以自由地访问(读和写)一个父类(parent)的所有参数,和上述被声明父类(parent)的所有的局部变量. 这是非常强大的.它通常允许拆分长的例行程序(routine)到几个小的例行程序中,而不需要很多努力 (因为你不必传递参数中所有的信息). 注意不要过度使用这个特征—如果一些嵌套的函数使用(甚至更改)父类(parent)的相同变量, 代码可能很难理解(follow).

这两个示例是相等的:

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

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

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

另一个版本,在这里我们让局部(嵌套)例行程序(routine)Square 来直接访问 I:

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

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

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

局部(嵌套)例行程序(routine)可以转到任何深度—这意味着你可以在其它的局部(嵌套)例行程序(routine)内定义一个局部(嵌套)例行程序(routine).所以你可以转到偏远地区(但是请不要转到

太偏远地区,不然代码将成为不可读:).

7.2. Callbacks（回调）(亦称 events（事件），亦称 pointers to functions（指针到函数），亦称 procedural variables（过程变量）)

它们允许间接地调用一个函数,通过一个变量.变量可以在运行时被分配到点到任何带有匹配参数类型和返回类型的函数.

callback(回调)可以是:

- 正常的,这意味着它可以点到任何正常的例行程序(routine)(不是一个方法,不是局部(嵌套)).

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

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

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

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

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

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

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

- 一个方法: 在尾部带有 of object 的声明.

```

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

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

  TMyClass = class
    CurrentValue: Integer;
    procedure Add(const A: Integer);
    procedure Multiply(const A: Integer);
    procedure ProcessTheList(const M: TMyMethod);
  end;

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

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

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

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

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

```

end.

注意你 不能 像方法一样传递全局 **procedures**(过程)/函数.它们是不兼容的.如果你不得不提供一个 **of object callback**(回调),除了不应该创建一个假程序类实例外,你可以像方法一样传递 类方法.

```
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;
```

可惜,你需要写怪模样的 @TMyClass(nil).Add 代替合理的 @TMyClass.Add.

- 一个(可能地)局部(嵌套)例行程序(routine):在尾部带有 **is nested** 声明,并确保对代码使用 `{ $modeswitch nestedprocvars }` 指令.它们与 **These go hand-in-hand with** 局部(嵌套)例行程序(routines)处于并进的状态.

7.3. 泛型(Generics)

任何的现代语言的一个强大的特征。一些事(通常,属于一个类)的定义可以用其它的类型参数化。最典型的示例是,当你需要创建一个容器(container)(一个列表,字典,树,图表...)时: 你可以定义类型 *T* 的一个列表,然后指定(*specialize*)它来立即获得整型的一个列表,字符串的一个列表, *TMyRecord* 的一个列表,等等.

在 Pascal 中的泛型更像在 C++中泛型的实现。这意味着在专门化 (specialization) 时它们是“扩展的”, 有一点像宏(但是比宏更安全; 例如, 标识符在泛型定义事被分解, 而不是在专门化 (specialization) 时, 因此当专门化 (specializing) 泛型时, 你不能“注入”任何的异常行为)。事实上,这意味着它们是非常快的(对每个详细的 (particular) 类型可以被最优化), 并且与任何大小的类型一起工作。当专门化 (specializing) 泛型时, 你可以使用一个基本的 (primitive) 类型 (整型, 浮点), 一个记录, 一个类。

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

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

procedure TMyCalculator.Add(const A: T);
```

```

begin
  Value := Value + A;
end;

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

var
  FloatCalc: TMyFloatCalculator;
  StringCalc: TMyStringCalculator;
begin
  FloatCalc := TMyFloatCalculator.Create;
  try
    FloatCalc.Add(3.14);
    FloatCalc.Add(1);
    Writeln('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.

```

泛型不是限定到类，你也可以有泛型函数和过程（procedures）：

```

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

```

关于重要的使用泛型的标准类型，参考[容器\(lists\(列表\),dictionaries\(词典\)\)使用泛型](#)。

7.4. Overloading（重载）

方法(全局函数和 `procedures`（过程）)带有相同的名称是被允许的，只要它们有不同的参数。在编译时，编译器检查出你想使用的一个，知道你传递的参数。

在默认情况下，重载使用 `FPC` 途径（`approach`），这意味着在给定命名空间（一个类或一个单元）中的所有的方法是相等的，并隐藏在命名空间中其它的带有优先级较低的方法。例如，如果你定义一个带有方法 `Foo(Integer)`和 `Foo(string)`的类，它继承自一个带有方法 `Foo(Float)`的类，那么，你新类的用户将不能容易地(它们仍然能 --- 如果它们类型强制转换类为它的原型（`ancestor`）类型)访问方法。为克服这个问题，使用 `overload` 关键字。

7.5. 预处理程序

你可以使用简单的预处理程序指令，对于

- 条件编译(代码依赖于平台，或一些自定义开关（`switches`）),
- 来包含在其它文件中一个文件，
- 你也可以使用无参数宏。

注意，带有参数的宏是不被允许的。通常，你应该避免使用预处理的原料（`stuff`）... 除非它真是合乎情理。预处理发生在语法分析前，这意味着你可以"打断"`Pascal` 语言的正常语法。这是一个强有力的，但也有些令人厌恶的特征。

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

{ $ifdef FPC }
{ This is only defined when compiled by FPC, not other compilers (like Delphi). }
procedure Foo;
{ $endif }

{ Define a NewLine constant. Here you can see how the normal syntax of Pascal
is "broken" by preprocessor directives. When you compile on Unix
(includes Linux, Android, Mac OS X), the compiler sees this:

const NewLine = #10;

When you compile on Windows, the compiler sees this:

const NewLine = #13#10;

On other operating systems, the code will fail to compile,
because a compiler sees this:

const NewLine = ;
```

It's a **good** thing that the compilation fails in this case -- if you will have to port the program to an OS that is not Unix, not Windows,

you will be reminded by a compiler to choose the newline convention on that system. }

```
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 is just a shortcut for $include
{$I some_other_file.inc}

end.
```

Include 文件通常有 `.inc` 扩展名，并且被用于两个目的：

- **Include** 文件可能仅包含其它编译器指令，这"配置"你的源文件代码。例如，你可以创建一个带有这些内容的文件 `myconfig.inc`：

```
{$mode objfpc}
{$H+}
{$J-}
{$modeswitch advancedrecords}
{$ifndef VER3}
  {$error This code can only be compiled using FPC version at least 3.x.}
{$endif}
```

现在你可以在你所有的源文件中使用 `{ $I myconfig.inc }` 包含这个文件。

- 其它常见的用法是分裂一个大的单元到一些文件中，直到语言规则被过度使用为止，仍然保持它为一个单独的单元。不要过度使用这个技巧—你的第一个直觉应该是来分裂一个单个单元到多个单元，而不是分裂一个单个单元到多个 **include**（包含）文件。永不减少，这是一个有用的技巧。

1. 它允许来避免"骤增"单元的数字，然而仍然保持你的源文件代码文件简短。例如，有一个带有"通常使用的 UI 控件"的单个的单元比创建一个用于每个 UI 控件类的单元可能会更好，因为后一种方法可能使平常的"uses"分句长(尽管一个典型的 UI 代码将依赖于一组 UI 类的几个)。但是，放置所有的这些 UI 类在一个单独的 myunit.pas 文件中可能使它成为一个长文件，难于找到正确的方法，因此，分裂它到多个 include（包含）文件能够可以理解。
2. 它允许有一个带有平台独立容易实现（implementation）的跨平台单元接口（interface）。基本上你可以做

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

有时，比写一个带有很多{\$ifdef UNIX}的长代码更好，{\$ifdef MSWINDOWS}与一般的代码(变量声明，程序（routine）implementation（实施）)混合。这种方式代码更可读。你甚至可以更积极地使用这个技巧，通过使用 FPC 的命令行选项-Fi 来包含一些仅对具体指定平台的子目录。然后你可以有很多 include（包含）文件{\$I my platform_specific_implementation.inc}的版本，你简单的 include（包含）它们，让编译器找到正确的版本。

7.6.记录

Record 只是其它变量的一个容器。它像一个大量简化的类: 没有继承或虚拟方法。它像在类 C 语言中的一个结构。

如果你使用{\$modeswitch advancedrecords}指令，记录能有方法和可视化说明符（specifiers）。

通常，可获得的类和不打破一个记录的简单可预测的存储器布局的语言特征是可能的。

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

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.WriteLineDescription;
end.

```

在 modern Object Pascal 中，你的第一直觉应该是来设计一个类，而不是一个记录——因为类有大量的特征，像构造函数和继承。

但是，当你需要速度或可预见的存储器布局时，记录仍然非常有用：

- 记录没有一些构造函数或解析函数。你仅定义一个记录类型的一个变量。它在开始处（除自动管理类型外，像字符串；它们被保证为初始化为空，并且最终来释放参考（reference）计数）有未定义内容（存储器垃圾）。所以，当处理记录时，你不得不更小心，但是它给予你一些性能增加。
- 在存储器中，记录的数组是令人满意地线性的，所以，它们是缓存友好的。
- 在一些情况中，记录（大小，字段之间的填充）的存储器布局被清晰地定义：当你要求 *C 布局*，或当你使用 *packed*（包装）记录时。这是有用的：
 - 与用其它编程语言写的库通信，当它们阐述（*expose*）一个基于记录的 API 时，
 - 读和写二进制文件，
 - 做令人厌恶的（*dirty*）低级技巧（像不安全的定型（*typecasting*）一个类型到另一个，意识到它们的存储器表现形式）。
- 记录也可以有 *case* 部分，这像联合作业在 C 类语言中。它们允许来对待相同的存储器块作为一个不同的类型，依赖于你的需要。像这样，在一些情况下，这给予更高的存储器效率。并且它给予令人厌恶的（*dirty*），低级不安全技巧：)

7.7. 旧样式对象

在过去，Turbo Pascal 引入其它语法，像使用 *object* 关键字的类功能。它有点介于一个记录和一个现代类之间概念的融合。

- 旧样式对象可以被分配/释放，并且在这操作期间，你可以调用它们的构造函数/析构函数。
- 但是，它们也可以被简单地声明和使用，像记录（*records*）。一个简单的记录或对象类型不是一个到其它事情的引用（指针），它不过是数据。这使它们对小数据舒适，在这里调用分配/释放可能会引起麻烦的。
- 旧样式对象提供继承和虚拟（*virtual*）方法，不过与来自现代类相比有小的不同。注意——如果你尝试来使用一个对象，而不调用它的构造函数，并且该对象有虚拟（*virtual*）方法，*不好的事情 things* 将会发生。

在大多数情况下，反对使用旧样式对象。现代类 *p* 提供更多的功能。并且当需要时，记录（包括 *高级记录*）能被用于执行（*performance*）。这些概念通常比旧样式对象更好

7.8. 指针

你可以创建一个 *指针* 到任意类型。指针到类型 *TMyRecord* 被声明为 *^TMyRecord*，并按照惯例被称为 *PMyRecord*。这是一个使用记录的整数链的一个传统示例：

```

type
  PMyRecord = ^TMyRecord;

```



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

注意，这定义是递归的(recursive)(类型 PMyRecord 是使用类型 TMyRecord 定义的, 而 TMyRecord 是使用类型 PMyRecord 定义的). 只要它在相同的类型语句块中将被解决，定义一个指针类型到一个 *尚未定义的类型*是被允许的。

你可以使用 New / Dispose 方法来分配和释放指针,或者使用(更低级，非类型安全的)GetMem / FreeMem 方法. 你间接引用指针(来访问 *指向的*原材料(stuff)，通过)你追加的^操作符.为执行逆运算，来获取一个存在值的一个指针,你使用@运算符前缀它。

也有一个未类型化的(untyped)指针类型,类似于在 C 类语言中的 void*.它是完全不安全的，可能被类型化(typecasted)到任意其它指针类型。

切记，事实上，一个类实例也是一个指针,尽管它不需要一些 ^ 或 @ 运算符来使用它. 一个使用类的链接表当然是合理的，它可以简单地是这样：

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

7.9.运算符(Operator)重载

你可以重载很多语言运算符的意义。例如，来给予你自定义类型的加法和乘法。像这样：

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

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

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

你也可以重载关于类的运算符.因为你通常在运算符函数中创建你的类的新的实例,调用者必需记着释放结果。

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

type
  TMyClass = class
    MyInt: Integer;
  end;

operator* (const C1, C2: TMyClass): TMyClass;
```

```
begin
  Result := TMyClass.Create;
  Result.MyInt := C1.MyInt * C2.MyInt;
end;
```

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

你也可以重载关于记录的运算符. 对于类来说,这通常比重载类它们更容易, 因为调用者不必须处理那时的存储器管理器.

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

type
  TMyRecord = record
    MyInt: Integer;
  end;

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

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

对于记录,建议在记录内部中使用 `{ $modeswitch advancedrecords }` 和重载运算符作为类运算符. 这允许使用泛型(`generic`)类,依赖于带有这些记录的一些运算符的存在(像 `TFPGList`,依赖于可用的相等的运算符). 否则,一个运算符(`operator`) (而不是在记录中)的"全局"定义可能不被找到(因为在实施(`implements`)`TFPGList` 的代码处不是可找到的), 并且你不能专门研究一个列表,像 `specialize TFPGList<TMyRecord>`.

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

```

{$modeswitch advancedrecords}
uses SysUtils, FGL;

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

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

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

type
  TMyRecordList = specialize TFPGList<TMyRecord>;

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

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

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

```

8. 高级的类特征

8.1. private（私有的）和 strict private（绝对的私有的）

Private(私有的) 可视性说明符(specifier)意味着, 在这个类的外部, 字段(或方法)不是可访问的。但是它允许一个例外: 所有定义在在相同的单元中的代码可以打破这个, 并且访问 private(私有的)字段和方法。一个 C++程序员可能会说, 在 Pascal 中, *所有在一单个单元中的类是朋友*。这常常是有用的, 并不打破你的封装(encapsulation), 因为它被限制到一个单元中。然而, 如果你创建较大的单元, 带有很大类(这些不与彼此紧密地合成一体), 使用 strict private(绝对的私有的) 是更安全的。这意味着, 在一段时间内 — 在这个类的外部, 字段(或方法) 不是可访问的.没有异常。

在一个类似的方法中, 这有 protected(受保护的) 可视性(在一个相同的单元中,对衍生物,或朋友是可视的) 和 strict protected(绝对的受保护的) (一段时间,对衍生物是可视的)。

8.2. 在类和嵌套的(nested)类的内部更多原料(stuff)

你可以打开一个类中的常量(const)或类型(type)的一部分.以这种方式,你甚至可以在一个类中定义一个类.这个可视性说明符一如既往地工作,尤其是嵌套的(nested)类可以被私有的(private)(对外部世界不可视),这常常的有用的。

注意,在一个常量或类型后声明一个字段,你将需要打开一个 var 语句块。

```
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;

{ note that method definition is prefixed with
  "TMyClass.TInternalClass" below. }
procedure TMyClass.TInternalClass.DoSomething;
begin
end;

```

8.3. 类方法

你能调用的这些方法有一个类参考(TMyClass), 不一定有一个类实例.

```

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;

```

注意,它们可以是虚拟的(virtual) — 它是合乎情理的,当与[类引用\(references\)](#)联合时,有时是非常有用的.

类方法也能通过[可视化说明符](#)限制,像 private 或 protected. 恰好像常规方法.

注意,当以一种正常的行为(MyInstance := TMyClass.Create(...);)调用时, 一个构造函数总是表现得像一个类方法. 尽管它也可能从类自身的内部调用一个构造函数, 像一个正常的方法, 并随后表现得像一个正常的方法. 这是一个有用的特色来"链接" 构造函数, 当一个构造函数 (例如, 重载以获取一个整型参数) 做相同的工作时, 并随后调用其它的构造函数 (例如, 无参数).

8.4. 类引用 (Class reference)

类引用允许你在运行时来选择类, 例如, 在编译时来调用一个类方法或不知道准确的类的构造函数. 它是一个被声明为 class of TMyClass 的类型.

```

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);

  // In addition, using class references, you can also do this:

  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;

```

类引用可以与虚拟类方法结合。这给予如同使用带有虚拟方法类的相同的效果——要执行的实际方法在运行时确定。

```

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;

```

```

end;

TMyClassRef = class of TMyClass;

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

  ClassRef := TMyClass2;
  ClassRef.DoSomething;

  { And this will cause an exception at runtime,
    since DoSomething is abstract in TMyClass. }
  ClassRef := TMyClass;
  ClassRef.DoSomething;
end;

```

如果你有一个实例，并且你想获得实例的类(不是声明的类，但是使用在它的构造函数处的最终衍生物类)一个引用，你可以使用 `ClassType` 属性。`ClassType` 的声明的类型是 `TClass`，其代表 `TObject` 的类。当你知道这个实例比 `TObject` 更具体指定类似时，你经常地安全地类型转换它到一些更具体指定的类型。

特别是，你可以使用 `ClassType` 引用来调用虚拟的方法，包含虚拟的构造函数。这允许你创建一个方法，像 `Clone`，构造 *当前对象的精确的运行时的一个实例*。你可以与 [Cloning: TPersistent.Assign](#) 一起结合它来拥有一个方法，该方法返回当前实例的一个新的构造的 `clone`。记住，它仅当你的类的构造函数是虚拟的时工作。例如它可以与标准的 `TComponent` 衍生物一起使用，尽管它们全部必需重写 `TComponent.Create(AOwner: TComponent)` 虚拟的构造函数。

```

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
  // This would always create an instance of exactly TMyClass:
  //Result := TMyClass.Create(AOwner);
  // This can potentially create an instance of TMyClass descendant:
  Result := TMyClassRef(ClassType).Create(AOwner);
  Result.Assign(Self);
end;

```

8.5. 静态类方法

为理解 *静态类方法* (*static class methods*)，你不得不理解 *正常类方法* (*normal class methods*) (在先前部分被描述) 是如何工作的。在内部，*正常类方法* 接收它们类(它通过一个隐藏的，不明显地添加方法的第一个参数)的一个 *类引用*。这个类引用甚至能使用在类方法内部的 *Self* 关键字被明确地访问。通常地，它是一个好东西：这个类引用允许你来调用 *虚拟的类方法* (通过该类的 *虚拟的类方法*)。

虽然这是极好的，当分配到一个 *全局 procedure* (过程) 指针时，它使 *正常的类方法* 不兼容。即，这将被不编译：

```
{ $mode objfpc } { $H+ } { $J- }
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
  // Error: TMyClass.Foo not compatible with TMyCallback
  Callback := @TMyClass(nil).Foo;
end.
```

注意	在 <i>Delphi</i> 模式中，你能够使写 <code>TMyClass.Foo</code> ，代替在上例中的一个危险的 <code>TMyClass(nil).Foo</code> 。不可否认， <code>TMyClass.Foo</code> 看起来更简洁，并且通过编译器更好地检查。使用 <code>TMyClass(nil).Foo</code> 是一个非法侵入... 不幸地，在 <i>ObjFpc</i> 模式中，在这边书提出的是必需的 (现在) 在任何情况下，分配 <code>MyClass.Foo</code> 到上面的回调可能在 <i>Delphi</i> 模式下 <i>仍然失败</i> ，对于完全相同的原因
----	--

上面的示例未能编译，因为回调与方法 `Foo` 是不兼容的。它是不兼容的，因为内部的类方法有特殊隐藏的 *implicit* 参数来传递一个类引用。

一种修复上面的示例方法是更改 `TMyCallback` 的定义。如果它是一个方法回调，它将工作，声明为 `TMyCallback = procedure (A: Integer) of object;`。但是有时，它不是可取的。

这里结合 *static* (静态) 类方法。实质上，它只是一个全局 *procedure* (过程) / 函数，但是它的命名空间被限制在类的内部。它 *确实* 没有隐式的 (*implicit*) 类引用 (因此，它 *不能是虚拟的*，并且它 *不能调用虚拟的类方法*)。在好的一面上，它与普通的 (非对象) 回调兼容，所以这将工作：

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

```

8.6. 类属性和变量

一个类属性是一个能通过一个类引用(它不需要一个类实例)访问的属性。

它是非常易懂的一个常规属性 (查看[属性](#))的类比。对于一个类属性，你定义一个 *getter* 和/或一个 *setter*。它们可以引用一个类变量或一个静态类方法。

一个类变量，你猜对了，像一个常规的字段，但是你不需要一个类实例来访问它。实质上，它恰好像一个全局变量，但是命名空间被限制到包含 (containing) 类。它可以被声明在类的 `class var` 部分。或者，它可以通过下面的带有关键字 `static` 的正常字段的定义被声明。一个静态类方法只是一个全局 `procedure` (过程)/函数，但是命名空间被限制到包含 (containing) 类。更多关于静态类方法在上面的部分中，查看[静态类方法](#)。

```

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

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.
```

8.7. 类助手 (helpers)

方法只是在一个类内部的一个 `procedure` (过程) 或函数。源自类的外部, 你使用一个特殊的语法 `MyInstance.MyMethod(...)` 调用它。在一段时间后, 你逐渐变得习惯于思考: 我是否想在实例 `X` 上执行动作 *Action*, 我写 `X.Action(...)`。

但是有时, 你需要实施 (implement) 一些东西, 这些东西在概念上是一个在类 `TMyClass` 上的动作, 而不修改 `TMyClass` 源文件代码。有时它是因为它不是你的源文件代码, 并且你不想更改它。有时它是因为依赖关系 — 添加一个方法, 像 `Render` 到一个类像 `TMy3DObject` 看起来像一个易懂的方法, 但是, 也许类 `TMy3DObject` 的基本实施 (implementation) 应该与 `rendering` (渲染) 代码保持独立? 它可能更好地"增强"一个现存的类, 来对它添加功能, 而不更改它的源文件代码。

简单完成它方法是接着创建一个全局 `procedure` (过程), 取走 `TMy3DObject` 一个的实例作为它的第一个参数。

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

这完美地工作, 但是, 缺点是调用它看起来有一点令人不快 (ugly)。在你通常调用动作期间, 像 `X.Action(...)`, 在这种情况下, 你不得不调用它们, 像 `Render(X, ...)`。能够仅仅写 `X.Render(...)` 可能是妙极的, 即使当 `Render` 没有实施 (implemented) 在的相同的单元时, 像 `TMy3DObject` 一样。

这是你使用类助手的地方。这仅是来实施 (implement) 在给定类上的控制 `procedures` (过程) / 函数的一种方法, 并且它们像方法一样被调用, 但事实上不是正常的方法 — 它们被添加在 `TMy3DObject` definition 定义的外部。

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

procedure TMy3DObjectHelper.Render(const Color: TColor);
var
  I: Integer;
begin
  // note that we access ShapesCount, Shape without any qualifiers here
  for I := 0 to ShapesCount - 1 do
    RenderMesh(Shape[I].Mesh, Color);
```

end;

注意

更常规的概念是“类型助手 (helpers)”。你甚至可以添加方法到原始的类型使用它们，像整形数或枚举。你也可以添加“记录助手 (helpers)”到(你猜对了...)记录。查看 <http://lists.freepascal.org/fpc-announce/2013-February/000587.html>。

8.8. 虚拟构造函数，析构函数

析构函数名称总是 `Destroy`，它是虚拟的(尽管你可以在编译时调用它而没有必要知道准确的类)和无参数的。

析构函数名称是按照惯例 `Create` (创建)。

你可以更改这种名称，不过注意这点 — 如果你定义 `CreateM`，也总是重新定义名称 `Create`，否则，用户仍能访问原型 (ancestor) 的构造函数 `Create`，绕过你的 `CreateMy` 构造函数。

在基础 `TObject` 中，在创建你能自由更改参数的衍生物期间，它不是虚拟的。新的构造函数将隐藏在原型 (注意：不要在这里放置 `overload`，除非你想破坏它) 中的构造函数。

在 `TComponent` 衍生物中，你应该重写它 `constructor (构造函数) Create(AOwner:`

`TComponent)`。对于流功能，在编译时创建一个类而没有必要知道它的类型，有虚拟构造函数是非常有用的(查看下面的“类引用”)。

8.9. 在构造函数中的一个异常

在一个构造函数期间，如果一个异常出现会发生什么？这一行

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

在这种情况下，不执行到最后，`X` 不能被分配，那么，谁将在一个不完全地构建的类后被清理？

Object Pascal 的解决方案是，万一一个异常出现在一个构造函数中，那么析构函数被调用。

这是为什么 *你的析构函数必需健壮* 的一个原因，这意味着它应该在任何情况下工作，甚至在一个部分地创建的类的实例上。通常，如果你安全地释放所有的东西，像通过 `FreeAndNil` 是容易的。

我们也不得不依赖于这些情况，*在构造函数被执行前，类的内存被保证正确的调到零*。所以我们知道它在开头，所有的类引用是 `nil`。所有的整型是 `0`，等等。

因此下面没有任何内存泄漏的工作：

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

```
uses SysUtils;
```

```
type
```

```
  TGun = class
```

```
  end;
```

```
  TPlayer = class
```

```
    Gun1, Gun2: TGun;
```

```
    constructor Create;
```

```
    destructor Destroy; override;
```

```
  end;
```

```

constructor TPlayer.Create;
begin
    inherited;
    Gun1 := TGun.Create;
    raise Exception.Create('Raising an exception from constructor!');
    Gun2 := TGun.Create;
end;

destructor TPlayer.Destroy;
begin
    { in case since the constructor crashed, we can
      have Gun1 <> nil and Gun2 = nil now. Deal with it.
      ...Actually, in this case, FreeAndNil deals with it without
      any additional effort on our side, because FreeAndNil checks
      whether the instance is nil before calling it's destructor. }
    FreeAndNil(Gun1);
    FreeAndNil(Gun2);
    inherited;
end;

begin
    try
        TPlayer.Create;
    except
        on E: Exception do
            Writeln('Caught ' + E.ClassName + ': ' + E.Message);
        end;
    end.
end.

```

9. 接口

9.1. Bare (CORBA) 接口

一个接口声明一个 API，很像一个类，但是它不定义实施（implementation）。一个类可以实施（implement）很多接口，但是它仅能有一个原型（ancestor）类。

你可以强制转换（cast）一个类到任何它支持的接口，然后 *通过这个接口调用方法*。这允许以统一的样式处理类，不从彼此衍生，但是仍然共享一些常见的功能。当一个简单的类继承不够用时是有用的。

在 Object Pascal 中 CORBA 接口的工作非常像在 Java 中的接口

(<https://docs.oracle.com/javase/tutorial/java/concepts/interface.html>)或在 C# 中的接口 (<https://msdn.microsoft.com/en-us/library/ms173156.aspx>)。

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

uses SysUtils, Classes;

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

  TMyClass1 = class(IMyInterface)
    procedure Shoot;
  end;

  TMyClass2 = class(IMyInterface)
    procedure Shoot;
  end;

  TMyClass3 = class
    procedure Shoot;
  end;

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

procedure TMyClass2.Shoot;
begin
  Writeln('TMyClass2.Shoot');
end;
```

```

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

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

```

9.2. CORBA 和接口的 COM 类型

为什么(上面提出的)接口被称为"CORBA"?

名称 **CORBA** 是不恰当的。一个更好的名称应该是 **bare 接口**。这些接口是一种"纯语言特色"。当你想强制转换各种各样的类为相同的接口时使用它们，因为它们共享一些常见的 API。

然而这些接口的类型可以与 *CORBA* (公共对象请求代理体系结构 *Common Object*

Request Broker Architecture) 技术一起被使用(查看 https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture)，它们不以任何方式相关联到这个技术。

需要声明 `{$interfaces corba}` 吗？

是的，因为在默认情况下，你创建 *COM* 接口。这可以通过表达 `{$interfaces com}` 而被明确地公布，但是它通常不需要，因为它是默认公布。

并且，我不建议使用 *COM* 接口，特别是你正在寻找一些等同于来自其它编程语言的接口。在 *Pascal* 中 *CORBA* 接口正是你期待的东西，如果你正在寻找一些等同于在 *C#* 和 *Java* 中的接口。与此同时，*COM* 接口引入你可能不想要的附加特色。

注意，这个 `{$interfaces xxx}` 声明仅影响没有任何明确的原型(仅关键字 `interface`，不是 `interface(ISomeAncestor)`)的接口。当一个接口有一个原型时，它有像原型一样的相同类，不理睬 `{$interfaces xxx}` 声明。

COM 接口是什么？

COM 接口等同于一个衍生自一个特殊 *IUnknown* 接口的接口。衍生自 *IUnknown*：

- 你的类需要定义 `_AddRef` 和 `_ReleaseRef` 方法。这些方法的正确实施 (implementation) 可以使用引用计数 (the reference-counting) 管理你的对象的生命周期。
- 添加 `QueryInterface` 方法。
- 允许与 *COM* (组件对象模型 (Component Object Model)) 技术相互作用。

为什么不建议你使用 *COM* 接口？

因为 *COM* 接口"缠住"两个特色，在我看来这应该是不相关的(正交的 (orthogonal))：*多重继承和引用计数*。其它编程语言为这两种特色精确地使用独立的概念。

为清晰易懂：*引用计数 (reference-counting)*，提供一个自动内存管理器(在简单的情况下，例如，无循环)，是一个非常有用的概念。但是，在我看来，用接口 (instead of making them orthogonal features) 缠住这个特色不是清晰易懂的。它当然不匹配我使用的具体情况。

- 有时，我想强制转换我的(其它不相关的)类到一个常用的接口。
- 有时，我想使用引用计数方法管理内存。
- 可能，将来我想与 *COM* 技术相互作用。

但是这些全部是独立的，不相关的需要。在我看来，在一个单独的语言特色中缠住它们是有相反作用。它确实引起实际的问题：

- 如果你需要 *强制转换类到一个常见的接口 API* 的特色，但是我不需要引用计数机制(我想手动释放对象)，那么 COM 接口是有问题的。甚至当引用计数被通过一个特殊的 `_AddRef` 和 `_ReleaseRef` 实施 (implementation) 禁用时，在你已经释放类实例后，你仍然需要注意一个未曾临时挂起的接口引用。关于它的更多细节在下一部分。
- 如果你需要 *引用计数* 的特色，但是我不需要一个接口层次结构来表示一些不同于类层次结构的东西，那么我不得不在接口中重复 (duplicate) 我的类。以此方式为每个类创建一个简单的接口。这是适得其反的。我更喜欢有 *智能指针* 作为一个单独的语言特色，而不是与接口 (并且幸好，它来到了：) 缠住在一起。

这是为什么我建议在所有现代代码用接口处理中使用 *CORBA* 样式接口，和 `{$interfaces corba}` 指令的原因。

*如果你在同一时间中仅需要"引用计数"和"多重继承"，那么使用 COM 接口。*此外，Delphi 现在仅有 COM 接口，所以，如果你的代码必需与 Delphi 兼容，你需要使用 COM 接口。

我们能使用接口持有 (have) 引用计数吗？

能。仅添加 `_AddRef` / `_ReleaseRef` 方法。这里不需要从 `IUnknown` 接口衍生。然而在大多数情况下，如果你想使用你的接口引用计数，你也可以只使用 COM 接口。

9.3.接口 GUIDs (全球唯一的标志符)

GUIDs 是看似随机的字符 `[{ABCD1234-...}]`，你可以看到它们被放置在每个接口的定义处。是的，它们只是随机的。令人失望的是，它们是必需的。

如果你不打算与像 *COM* 或 *CORBA* 一样的通信技术集成，GUIDs 没有意义。但是对于实施 (implementation) 原因，它们是必需的。不要被编译器糊弄，令人失望的是编译器允许你不带有 GUIDs 来声明接口。

不带有(唯一的) GUIDs，你的接口将被等同是操作符对待。实际上，如果你的类支持任何一个你的接口，它将返回 `true`。在这里，神奇的函数 `Supports(ObjectInstance, IMyInterface)` 表现稍微更好一点，因为它拒绝编译不带有一个 GUID 的接口。在 FPC 3.0.0 时，对于 *CORBA* 和 *COM* 接口是正确的。

所以，为了慎重起见，你应该总是为你的接口声明一个 GUID。你可以使用 *Lazarus* GUID 生成器 (在编辑器中的快捷方式是 `Ctrl + Shift + G`)。或你可以使用一个在线服务，像

<https://www.guidgenerator.com/>。

或者，你可以为此使用 RTL 中的 `CreateGUID` 和 `GUIDToString` 函数写你自己的工具。查看下面的示例：

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

var
  MyGuid: TGUID;
begin
  Randomize;
```



```
CreateGUID(MyGuid);
Writeln(['" + GUIDToString(MyGuid) + "']);
end.
```

9.4. 引用计数(COM)接口

COM 接口带来两个附加特色:

1. 与 COM (来自 Windows 的一个技术, 也在 Unix 上通过 XPCOM 可用, 由 Mozilla 使用) 集成,
2. 引用计数(当所有的接口引用超出范围时, 为你提供自动销毁)。

当使用 COM 接口时, 你需要知道自动销毁机制和与 COM 技术的关系。

在实践中, 这意味着:

- 你的类需要实施 (implement) 一个神奇的 `_AddRef`, `_Release`, 和 `QueryInterface` 方法。或衍生自一些已经实施 (implements) 它们的东西。这些方法的一个特别的实施 (implementation) 可能事实上启用或禁用 COM 接口(尽管禁用它是有点危险—查看下一个重点)的引用计数特色。
 - 标准的类 `TInterfacedObject` 实施 (implements) 这些方法来启用引用计数。
 - 标准的类 `TComponent` 实施 (implements) 这些方法来禁用引用计数。在 **Castle Game Engine** 中, 为了这个目的, 我们给你附加的有用的原型 `TNonRefCountedInterfacedObject` 和 `TNonRefCountedInterfacedPersistent`, 查看 <https://github.com/castle-engine/castle-engine/blob/master/src/base/castleinterfaces.pas>。
- 当它可能被一些接口变量引用时, 你需要注意 释放类。因为接口是使用虚拟方法(因为它能被引用计数, 即使你非法侵入 `_AddRef` 方法, 而不非引用计数...)发布的, 你不能释放底层的对象实例, 只要一些借口变量可能指向它。查看, **FPC 手册** (<http://freepascal.org/docs-html/ref/refse47.html>)中的 "7.7 引用计数"。

使用 COM 接口的最安全的方法是

- 接受它们被引用计数的事实,
- 从 `acedObject` 衍生得到合适的类,
- 并避免使用类实例, 反而总是通过接口访问实例, 让引用计数管理存储单元分配。

这是这样的接口实例的一个示例:

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

uses SysUtils, Classes;

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

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

```

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

TMyClass3 = class(TInterfacedObject)
    procedure Shoot;
end;

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

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

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

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

var
    C1: IMyInterface; // COM takes care of destruction
    C2: IMyInterface; // COM takes care of destruction
    C3: TMyClass3;    // YOU have to take care of destruction
begin
    C1 := TMyClass1.Create as IMyInterface;
    C2 := TMyClass2.Create as IMyInterface;
    C3 := TMyClass3.Create;
    try
        UseThroughInterface(C1); // no need to use "as" operator
        UseThroughInterface(C2);
        if C3 is IMyInterface then
            UseThroughInterface(C3 as IMyInterface); // this will not execute
    finally
        { C1 and C2 variables go out of scope and will be auto-destroyed now.

```

```
In contrast, C3 is a class instance, not managed by an interface,  
and it has to be destroyed manually. }  
FreeAndNil(C3);  
end;  
end.
```

9.5. 禁用引用计数使用 COM 接口

如同在前面部分提到，你的类可以从 `TComponent` (或一个类似的类，像 `TNonRefCountedInterfacedObject` 和 `TNonRefCountedInterfacedPersistent`) 衍生，为 COM 界面禁用引用计算。这允许你使用 COM 接口，并且仍然可以手动释放类。

当一些接口变量可能引用它时，在这种情况下，你需要注意不释放类实例。记住，每个类型强制转换 `Cx` 为 `IMyInterface`，也创建一个临时的接口变量，甚至可能到当前的 `procedure`（过程）结尾时出现。因为这个原因，下面的示例使用一个 `UseInterfaces procedure`（过程），并且它释放在这个 `procedure`（过程）(当我们可以确保这个临时接口变量在范围外时)外部的类实例。

为避免这种混乱，通常，使用 CORBA 接口更好，如果你不想带有你的接口引用计数。

```
{ $mode objfpc } { $H+ } { $J- }  
{ $interfaces com }  
  
uses SysUtils, Classes;  
  
type  
  IMyInterface = interface  
    ['{3075FFCD-8EFB-4E98-B157-261448B8D92E}']  
    procedure Shoot;  
  end;  
  
  TMyClass1 = class(TComponent, IMyInterface)  
    procedure Shoot;  
  end;  
  
  TMyClass2 = class(TComponent, IMyInterface)  
    procedure Shoot;  
  end;  
  
  TMyClass3 = class(TComponent)  
    procedure Shoot;  
  end;  
  
procedure TMyClass1.Shoot;  
begin  
  Writeln('TMyClass1.Shoot');  
end;
```

```

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

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

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

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

procedure UseInterfaces;
begin
    if C1 is IMyInterface then
        //if Supports(C1, IMyInterface) then // equivalent to "is" check above
        UseThroughInterface(C1 as IMyInterface);
    if C2 is IMyInterface then
        UseThroughInterface(C2 as IMyInterface);
    if C3 is IMyInterface then
        UseThroughInterface(C3 as IMyInterface);
end;

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

```

9.6.不带有"as"操作符类型强制转换接口

这部分适用于 *CORBA* 和 *COM* 接口。

在运行时使用操作符强制转换到一个接口类型进行一次检查。考虑这行代码：

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

它适用于先前部分示例中的 *C1* , *C2* , *C3* 实例。如果执行, 在 *C3* 的情况下可能发生一个运行时错误, 它不实施 (implement) *IMyInterface* (但是我们在做强制转换前, 通过控制 *Cx* 是 *IMyInterface*, 避免该错误)。

你可以代替强制转换实例像一个隐式接口一样:

```
UseThroughInterface(Cx);
```

在这种情况下, 在编译时, 类型强制转换必需是有效的。所以这将对 *C1* 和 *C2* (它们被声明为实施 (implement) *IMyInterface* 的类)编译。但是它将不对 *C3* 编译。

本质上, 这种类型强制转换的样子和工作方式正像常规的类。究竟在哪里要求一个类 *TMyClass* 的一个实例, 你总是可以在一个变量哪里使用, 变量使用 *TMyClass* 的一个类或 *TMyClass* 衍生物声明。相同的规则适用于接口。在这样的情况下, 不需要任何显式类型强制转换。

同样, 一个相等的东西是

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

这也是一种在编译时必需是有效的类型强制转换。注意, 这种语法与类类型强制转换不一致。在类的情况下, 写 *TMyClass(C)*是一个不安全的, 不受限制的类型强制转换。在接口的情况下, 写 *IMyInterface(C)*是一种安全的, 敏捷的 (在编译时检查)类型强制转换。

10. 关于这个文档

版权 Michalis Kamburelis.

这篇文档的源文件代码是 Ascii 文档格式，在 <https://github.com/michaliskambi/modern-pascal-introduction>。对于修改和增加的建议，补丁和推送的要求，总是非常欢迎 :)，你可以联系我，通过 GitHub 或电子邮件 michalis.kambi@gmail.com。我的主页是

<https://michalis.ii.uni.wroc.pl/~michalis/>。这篇文档被链接在 *Castle Game Engine* 网站 <https://castle-engine.io/> 的 *Documentation*（文档）部分。

你可以自由地分发、甚至修改这篇文档，在相同的协议下，如 Wikipedia

<https://en.wikipedia.org/wiki/Wikipedia:Copyrights>：

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

感谢阅读！



山东世联环保科技开发有限公司

Shandong World United Environmental Protection Technology Development Co.,Ltd.

世联环保官方网站: <http://www.cnwut.com>

感谢 山东世联环保科技开发有限公司 资助 Lazarus 书籍的中文化翻译。

希望大家帮助与支持 山东世联环保科技开发有限公司 的发展，

为 世联环保 提供业务信息，进而支持 Lazarus 中文化发展！

