Modern Object Pascal Introduction for Programmers

Michalis Kamburelis

Table of Contents

1. Why

There are many books and resources about Pascal out there, but too many of them talk about the old Pascal, without classes, units or generics.

So I wrote this quick introduction to what I call **modern Object Pascal**. Most of the programmers using it don't really call it *"modern Object Pascal"*, we just call it *"our Pascal"*. But when introducing the language, I feel it's important to emphasize that it's a modern, object-oriented language. It evolved a **lot** since the old (Turbo) Pascal that many people learned in schools long time ago. Feature-wise, it's quite similar to C++ or Java or C#.

- It has all the modern features you expect — classes, units, interfaces, generics…
- It's compiled to a fast, native code,
- It's very type safe,
- High-level but can also be low-level if you need it to be.

It also has excellent, portable and open-source compiler called the *Free Pascal Compiler*, http://freepascal.org/ . And an accompanying IDE (editor, debugger, a library of visual components, form designer) called *Lazarus* http://lazarus.freepascal.org/ . Myself, I'm the creator of *Castle Game Engine*, https://castle-engine.sourceforge.io/ , which is a cool portable 3D and 2D game engine using this language to create games on many platforms (Windows, Linux, MacOSX, Android, iOS, web plugin).

This introduction is mostly directed at programmers who already have experience in other languages. We will not cover here the meanings of some universal concepts, like *"what is a class"*, we'll only show how to do them in Pascal.

2. Basics

2.1. "Hello world" program

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

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

This is a complete program that you can *compile* and *run*.

- If you use the command-line FPC, just create a new file `myprogram.lpr` and execute `fpc myprogram.lpr`.

- If you use *Lazarus*, create a new project (menu *Project → New Project → Simple Program*). Save it as `myprogram` and paste this source code as the main file. Compile using the menu item *Run → Compile*.
- This is a command-line program, so in either case — just run the compiled executable from the command-line.

The rest of this article talks about the Object Pascal language, so don't expect to see anything more fancy than the command-line stuff. If you want to see something cool, just create a new GUI project in *Lazarus* (*Project → New Project → Application*). Voila — a working GUI application, cross-platform, with native look everywhere, using a comfortable visual component library. The *Lazarus* and *Free Pascal Compiler* come with lots of ready units for networking, GUI, database, file formats (XML, json, images…), threading and everything else you may need. I already mentioned my cool *Castle Game Engine* earlier:)

2.2. Functions, procedures, primitive types

```
{$mode objfpc}{$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.
```

To return a value from a function, assign something to the magic `Result` variable. You can read and set the `Result` freely, just like a local variable.

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

You can also treat the function name (like `MyFunction` in example above) as the variable, to which you can assign. But I would discourage it in new code, as it looks "fishy" when used on the right side of the assignment expression. Just use `Result` always when you want to read or set the function result.

If you want to call the function itself recursively, you can of course do it. If you're calling a parameter-less function recursively, be sure to specify the parenthesis (even though in Pascal you can usually omit the parentheses for a parameter-less function), this makes a recursive call to a parameter-less function different from accessing this function's current result. Like this:

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

You can call `Exit` to end the execution of the procedure or function before it reaches the final `end;`. If you call parameter-less `Exit` in a function, it will return the last thing you set as `Result`. You can also use `Exit(X)` construct, to set the function result and exit **now** — this is just like `return X` construct in C-like languages.

```
function AddName(const ExistingNames, NewName: string): string;
begin
```

```
  if ExistingNames = '' then
    Exit(NewName);
  Result := ExistingNames + ', ' + NewName;
end;
```

2.3. Testing (if)

Use `if .. then` or `if .. then .. else` to run some code when some condition is satisfied. Unlike in the C-like languages, in Pascal you don't have to wrap the condition in parenthesis.

```
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;
```

The `else` is paired with the last `if`. So this works as you expect:

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

```
    AIsNonzeroButBIsZero;
```

While the example with nested `if` above is correct, it is often better to place the nested `if` inside a `begin` ... `end` block in such cases. This makes the code more obvious to the reader, and it will remain obvious even if you mess up the indentation. The improved version of the example is below. When you add or remove some `else` clause in the code below, it's obvious to which condition it will apply (to the `A` test or the `B` test), so it's less error-prone.

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

2.4. Logical, relational and bit-wise operators

The *logical operators* are called `and`, `or`, `not`, `xor`. Their meaning is probably obvious (search for *"exclusive or"* if you're unsure what *xor* does:). They take *boolean arguments*, and return a *boolean*. They can also act as *bit-wise operators* when both arguments are integer values, in which case they return an integer.

The *relational (comparison)* operators are `=`, `<>`, `>`, `<`, `<=`, `>=`. If you're accustomed to C-like languages, note that in Pascal you compare two values (check are they equal) using a single equality character `A = B` (unlike in C where you use `A == B`). The special *assignment* operator in Pascal is `:=`.

The *logical (or bit-wise) operators have a higher precedence than relational operators*. So you may need to use parenthesis around some expressions.

For example this is a compilation error:

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

The above fails to compile, because the compiler sees the bit-wise `and` inside: `(0 and B)`.

This is correct:

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

The *short-circuit evaluation* is used. Consider this expression:

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

- It's guaranteed that `MyFunction(X)` will be evaluated first.
- And if `MyFunction(X)` returns `false`, then the value of expression is known (the value of `false and whatever` is always `false`), and `MyOtherFunction(Y)` will not be executed at all.
- Analogous rule is for `or` expression. There, if the expression is known to be `true` (because the 1st operand is `true`), the 2nd operand is not evaluated.
- This is particularly useful when writing expressions like

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

  This will work OK, even when `A` is `nil`.

2.5. Testing single expression for multiple values (case)

If a different action should be executed depending on the value of some expression, then the `case .. of .. end` statement is useful.

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

```
end;
```

The `else` clause is optional. When no condition matches, and there's no `else`, then nothing happens.

In you come from C-like languages, and compare this with `switch` statement in these languages, you will notice that there is no automatic *fall-through*. This is a deliberate blessing in Pascal. You don't have to remember to place `break` instructions. In every execution, *at most one* branch of the `case` is executed, that's it.

2.6. Enumerated and ordinal types and sets and constant-length arrays

Enumerated type in Pascal is a very nice, opaque type. You will probably use it much more often than enums in other languages:)

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

The convention is to prefix the enum names with a two-letter shortcut of type name, hence `ak` = shortcut for *"Animal Kind"*. This is a useful convention, since the enum names are in the unit (global) namespace. So by prefixing them with `ak` prefix, you minimize the chances of collisions with other identifiers.

The collisions in names are not a show-stopper. It's Ok for different units to define the same identifier. But it's a good idea to try to avoid the collisions anyway, to keep code simple to understand and grep.

You can avoid placing enum names in the global namespace by directive `{$scopedenums on}`. This means you will have to access them qualified by a type name, like `TAnimalKind.akDuck`. The need for `ak` prefix disappears in this situation, and you will probably just call the enums `Duck, Cat, Dog`. This is similar to C# enums.

The fact that enumerated type is *opaque* means that it cannot be just assigned to and from an integer. However, for special use, you can use `Ord(MyAnimalKind)` to forcefully convert enum to int, or typecast `TAnimalKind(MyInteger)` to forcefully convert int to enum. In the latter case, make sure to check first whether `MyInteger` is in good range (0 to `Ord(High(TAnimalKind))`).

Enumerated and ordinal types can be used as array indexes:

```
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;
```

They can also be used to create sets (a bit-fields internally):

```
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. Loops (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.
```

About the `repeat` and `while` loops:

It may seem that the difference between the `while` and `repeat` loops is "cosmetic" (you specify a negated condition, because in `while .. do` you tell it *when to continue*, but in `repeat .. until` you tell it *when to stop*). Actually there's another important difference: in case of `repeat`, *the condition is not checked at the beginning*. So the `repeat` loop always runs at least once.

About the `for I := …` loops:

The `for I := .. to .. do …` construction it similar to the C-like `for` loop. However, it's more constrained, as you cannot specify arbitrary actions/tests to control the loop iteration. This is strictly for iterating over a consecutive numbers (or other ordinal types). The only flexibility you have is that you can use `downto` instead of `to`, to make numbers go downward.

In exchange, it looks clean, and is very optimized in execution. In particular, *the expressions for the lower and higher bound are only calculated once*, before the loop starts.

Note that the value of the loop counter variable (`I` in this example) should be considered *undefined* after the loop has finished, due to possible optimizations. Accessing the value of `I` after the loop may cause a compiler warning. *Unless* you exit the loop prematurely by `Break` or `Exit`: in such case, the counter variable is guaranteed to retain the last value.

About the `for I in …` loops:

The `for I in .. do ..` is similar to `foreach` construct in many modern languages. It works intelligently on many built-in types:

- It can iterate over all values in the array (example above).
- It can iterate over all possible values of an enumerated type:

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

- It can iterate over all items included in the set:

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

- And it works on custom list types, generic or not, like `TObjectList` or `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.
```

We didn't yet explain the concept of classes, so the last example may not be obvious to you yet — just carry on, it will make sense later:)

2.8. Output, logging

To simply output strings in Pascal, use the `Write` or `Writeln` routine. The latter automatically adds a newline at the end.

This is a "magic" routine in Pascal. It takes a variable number of arguments and they can have any type. They are all converted to strings when displaying, with a special syntax to specify padding and number precision.

```
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);
```

To explicitly use newline in the string, use the `LineEnding` constant (from FPC RTL). (The *Castle Game Engine* defines also a shorter `NL` constant.) Pascal strings do not interpret any special backslash sequences, so writing

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

doesn't work like some of you would think. This will work:

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

or just this:

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

Note that this will only work in *console* applications. Make sure you have `{$apptype CONSOLE}` (and **not** `{$apptype GUI}`) defined in your main program file. On some operating systems it actually doesn't matter and will work always (Unix), but on some operating systems trying to write something from a GUI application is an error (Windows).

In the Castle Game Engine: use `WritelnLog` or `WritelnWarning`, never `Writeln`, to print debug information. They will be always directed to some useful output. On Unix, standard output. On Windows GUI application, log file. On Android, the *Android logging facility* (visible when you use `adb logcat`). The use of `Writeln` should be limited to the cases when you write a command-line application (like a 3D model converter / generator) and you know that the *standard output* is available.

2.9. Converting to a string

To convert an arbitrary number of arguments to a string (instead of just directly outputting them), you have a couple of options.

- You can convert particular types to strings using specialized functions like `IntToStr` and `FloatToStr`. Furthermore, you can concatenate strings in Pascal simply by adding them. So you can create a string like this: `'My int number is ' + IntToStr(MyInt) + ', and the value of Pi is ' + FloatToStr(Pi)`.
  - *Advantage*: Absolutely flexible. There are many `XxxToStr` overloaded versions and friends (like `FormatFloat`), covering many types.
  - *Another advantage*: Consistent with the reverse functions. To convert a string (for example, user input) back to an integer or float, you use `StrToInt`, `StrToFloat` and friends (like `StrToIntDef`).
  - *Disadvantage*: A long concatenation of many `XxxToStr` calls and strings doesn't look nice.

- The `Format` function, used like `Format('%d %f %s', [MyInt, MyFloat, MyString])`. This is like `sprintf` function in the C-like languages. It inserts the arguments into the placeholders in the pattern. The placeholders may use special syntax to influence formatting, e.g. `%.4f` results in a floating-point format with 4 digits after the decimal point.
    - *Advantage*: The separation of *pattern* string from *arguments* looks clean. If you need to change the pattern string without touching the arguments (e.g. when translating), you can do it easily.
    - *Another advantage*: No compiler magic. You can use the same syntax to pass any number of arguments of an arbitrary type in your own routines (declare parameter as an `array of const`). You can then pass these arguments downward to `Format`, or deconstruct the list of parameters and do anything you like with them.
    - *Disadvantage*: Compiler does not check whether the pattern matches the arguments. Using a wrong placeholder type will result in an exception at runtime (`EConvertError` exception, not anything nasty like a segmentation fault).
- `WriteStr(TargetString, …)` routine behaves much like `Write(…)`, except that the result is saved to the `TargetString`.
    - *Advantage*: It supports all the features of `Write`, including the special syntax for formatting like `Pi:1:4`.
    - *Disadvantage*: The special syntax for formatting is a "compiler magic", implemented specifically for routines like this. This is sometimes troublesome, e.g. you cannot create your own routine `MyStringFormatter(…)` that would also allow the special syntax like `Pi:1:4`. For this reason (and also because it wasn't implemented for a long time in major Pascal compilers), this construction is not very popular.

3. Units

Units allow you to group common stuff (anything that can be declared), for usage by other units and programs. They are equivalent to *modules* and *packages* in other languages. They have an interface section, where you declare what is available for other units and programs, and then the implementation. Save unit `MyUnit` as `myunit.pas` (lowercase with `.pas` extension).

```
{$mode objfpc}{$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.
```

Final programs are saved as `myprogram.lpr` files (`lpr` = Lazarus program file; in Delphi you would use `.dpr`). Note that other conventions are possible here, e.g. some projects just use `.pas` for main program file, some use `.pp` for units or programs. I advice using `.pas` for units and `.lpr` for FPC/Lazarus programs.

A program can use a unit by a `uses` keyword:

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

program MyProgram;

uses MyUnit;

begin
  Writeln(MyFunction('Note: '));
  MyProcedure(5);
end.
```

3.1. Units using each other

One unit can also use another unit. Another unit can be used in the interface section, or only in the implementation section. The former allows to define new public stuff (procedures, types…) on top of another unit's stuff. The latter is more limited (if you

use a unit only in the implementation section, you can use it's identifiers only in your implementation).

```
{$mode objfpc}{$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.
```

It is not allowed to have *circular unit dependencies in the interface*. That is, two units cannot use each other in the interface section. The reason is that in order to "understand" the interface section of a unit, the compiler must first "understand" all the units it uses in the interface section. Pascal language follows this rule strictly, and it allows a fast compilation and fully automatic detection on the compiler side *what units need to be recompiled*. There is no need to use complicated `Makefile` files for a simple task of compilation in Pascal, and there is no need to *recompile everything* just to make sure that all dependencies are updated correctly.

It is *OK to make a circular dependency between units when at least one "usage" is only in the implementation*. So it's OK for unit `A` to use unit `B` in the interface, and then unit `B` to use unit `A` in the implementation.

3.2. Qualifying identifiers with unit name

Different units may define the same identifier. To keep the code simple to read and search, you should usually avoid it, but it's not always possible. In such cases, the last unit on the `uses` clause "wins", which means that the identifiers it introduces hide the same identifiers introduced by earlier units.

You can always explicitly define a unit of a given identifier, by using it like `MyUnit.MyIdentifier`. This is the usual solution when the identifier you want to use from `MyUnit` is hidden by another unit. Of course you can also rearrange the order of units on your uses clause, although this can affect other declarations than the one you're trying to fix.

```
{$mode objfpc}{$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.
```

In case of units, remember that they have two `uses` clauses: one in the interface, and another one in the implementation. The rule *later units hide the stuff from earlier units* is applied here consistently, which means that *units used in the implementation section* can hide identifiers from units *used in the interface section*. However, remember that when reading the `interface` section, only the units used in the interface matter. This may create a confusing situation, where two seemingly-equal declarations are considered different by the compiler:

```
{$mode objfpc}{$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.
```

The unit `Graphics` (from Lazarus LCL) defines the `TColor` type. But the compiler will fail to compile the above unit, claiming that you don't implement a procedure `ShowColor` that matches the interface declaration. The problem is that unit `GoogleMapsEngine` also defines a `TColor` type. And it is used only in the `implementation` section, therefore it *shadows* the `TColor` definition only in the implementation. The equivalent version of the above unit, where the error is obvious, looks like this:

```
{$mode objfpc}{$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.
```

The solution is trivial in this case, just change the implementation to explicitly use `TColor` from `Graphics` unit. You could fix it also by moving the `GoogleMapsEngine` usage, to the interface section and earlier than `Graphics`, although this could result in other consequences in real-world cases, when `UnitUsingColors` would define more things.

```
{$mode objfpc}{$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. Exposing one unit identifiers from another

Sometimes you want to take an identifier from one unit, and *expose* it in a new unit. The end result should be that using the new unit will make the identifier available in the namespace.

Sometimes this is necessary to preserve backward compatibility with previous unit versions. Sometimes it’s nice to "hide" an internal unit this way.

This can be done by redefining the identifier in your new unit.

```
{$mode objfpc}{$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.
```

Note that this trick cannot be done as easily with global procedures, functions and variables. With procedures and functions, you could expose a constant pointer to a procedure in another unit (see Section 7.2, “Callbacks (aka events, aka pointers to functions, aka procedural variables)”), but that looks quite dirty.

The usual solution is then to create a trivial "wrapper" functions that underneath simply call the functions from the internal unit, passing the parameters and return values around.

To make this work with global variables, one can use global (unit-level) properties, see Section 4.3, “Properties”.

4. Classes

4.1. Basics

We have classes. At the basic level, a class is just a container for

- *fields* (which is fancy name for *"a variable inside a class"*),
- *methods* (which is fancy name for *"a procedure or function inside a class"*),
- and *properties* (which is a fancy syntax for something that looks like a field, but is in fact a pair of methods to *get* and *set* something; more in Section 4.3, “Properties”).
- Actually, there are more possibilities, described in Section 8.2, “More stuff inside classes and nested classes”.

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

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

4.2. Inheritance, is, as

We have inheritance and 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.
```

By default methods are not virtual, declare them with `virtual` to make them. Overrides must be marked with `override`, otherwise you will get a warning. To hide a method without overriding (usually you don't want to do this, unless you now what you're doing) use `reintroduce`.

To test the class of an instance at runtime, use the `is` operator. To typecast the instance to a specific class, use the `as` operator.

```
{$mode objfpc}{$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.
```

Instead of casting using `X as TMyClass`, you can also use the *unchecked* typecast `TMyClass(X)`. This is faster, but results in an undefined behavior if the `X` is not, in fact, a `TMyClass` descendant. So don't use the `TMyClass(X)` typecast, or use it only in a code where it's blindingly obvious that it's correct, for example right after testing with `is`:

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

4.3. Properties

Properties are a very nice *"syntax sugar"* to

1. Make something that looks like a field (can be read and set) but underneath is realized by calling a *getter* and *setter* methods. The typical usage is to perform some side-effect (e.g. redraw the screen) each time some value changes.
2. Make something that looks like a field, but is read-only. In effect, it's like a constant or a parameter-less function.

```
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;
```

Note that instead of specifying a method, you can also specify a field (typically a private field) to directly get or set. In the example above, the `Color` property uses a *setter*

method `SetColor`. But for getting the value, the `Color` property refers directly to the private field `FColor`. Directly referring to a field is faster than implementing trivial getter or setter methods (faster for you, and faster at execution).

When declaring a property you specify:

1. Whether it can be read, and how (by directly reading a field, or by using a "getter" method).
2. And, in a similar manner, whether it can be set, and how (by directly writing to a designated field, or by calling a "setter" method).

The compiler checks that the types and parameters of indicated fields and methods match with the property type. For example, to read an `Integer` property you have to either provide an `Integer` field, or a parameter-less method that returns an `Integer`.

Technically, for the compiler, the "getter" and "setter" methods are just normal methods and they can do absolutely anything (including side-effects or randomization). But it's a good convention to design properties to behave more-or-less like fields:

- The *getter* function should have no visible side-effects (e.g. it should not read some input from file / keyboard). It should be deterministic (no randomization, not even pseudo-randomization :). Reading a property many times should be valid, and return the same value, if nothing changed in-between.

  Note that it's OK for *getter* to have some *invisible* side-effect, for example to cache a value of some calculation (known to produce the same results for given instance), to return it faster next time. This is in fact one of the cool possibilities of a "getter" function.

- The *setter* function should always set the requested value, such that calling the *getter* yields it back. Do not reject invalid values silently in the "setter" (raise an exception if you must). Do not convert or scale the requested value. The idea is that after `MyClass.MyProperty := 123;` the programmer can expect that `MyClass.MyProperty = 123`.

- The *read-only properties* are often used to make some field read-only from the outside. Again, the good convention is to make it behave like a constant, at least constant for this object instance with this state. The value of the property should not change unexpectedly. *Make it a function, not a property, if using it has a side effect or returns something random.*

- The *"backing" field of a property is almost always private*, since the idea of a property is to encapsulate all outside access to it.
- It's technically possible to make *set-only properties*, but I have not yet seen a good example of such thing:)

> Properties can also be defined outside of class, at a unit level. They serve an analogous purpose then: look like a global variable, but are backed by a *getter* and *setter* routines.

4.4. Exceptions

We have exceptions. They can be caught with `try … except … end` clauses, and we have finally sections like `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.
```

Note that the `finally` clause is executed even if you exit the block using the `Exit` (from function / procedure / method) or `Break` or `Continue` (from loop body).

4.5. Visibility specifiers

As in most object-oriented languages, we have visibility specifiers to hide fields / methods / properties.

The basic visibility levels are:

`public`
: everyone can access it, including the code in other units.

`private`
: only accessible in this class.

`protected`
: only accessible in this class and descendants.

The explanation of `private` and `protected` visibility above is not precisely true. The code *in the same unit* can overcome their limits, and access the `private` and `protected` stuff freely. Sometimes this is a nice feature, allows you to implement tightly-connected classes. Use `strict private` or `strict protected` to secure your classes more tightly. See the Section 8.1, “private and strict private”.

By default, if you don’t specify the visibility, then the visibility of declared stuff is `public`. The exception is for classes compiled with `{$M+}`, or descendants of classes compiled with `{$M+}`, which includes all descendants of `TPersistent`, which also includes all descendants of `TComponent` (since `TComponent` descends from `TPersistent`). For them, the default visibility specifier is `published`, which is like `public`, but in addition the streaming system knows to handle this.

Not every field and property type is allowed in the `published` section (not every type can be streamed, and only classes can be streamed from simple fields). Just use `public` if you don’t care about streaming but want something available to all users.

4.6. Default ancestor

If you don’t declare the ancestor type, every `class` inherits from `TObject`.

4.7. Self

The special keyword `Self` can be used within the class implementation to explicitly refer to your own instance. It is equivalent to `this` from C++, Java and similar languages.

4.8. Calling inherited method

Within a method implementation, if you call another method, then by default you call the method of your own class. In the example code below, `TMyClass2.MyOtherMethod` calls `MyMethod`, which ends up calling `TMyClass2.MyMethod`.

```
{$mode objfpc}{$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.
```

If the method is not defined in a given class, then it calls the method of an ancestor class. In effect, when you call `MyMethod` on an instance of `TMyClass2`, then

- The compiler looks for `TMyClass2.MyMethod`.
- If not found, it looks for `TMyClass1.MyMethod`.
- If not found, it looks for `TObject.MyMethod`.
- if not found, then the compilation fails.

You can test it by commenting out the `TMyClass2.MyMethod` definition in the example above. In effect, `TMyClass1.MyMethod` will be called by `TMyClass2.MyOtherMethod`.

Sometimes, you don't want to call the method of your own class. You want to call the method of an ancestor (or ancestor's ancestor, and so on). To do this, add the keyword `inherited` before the call to `MyMethod`, like this:

```
inherited MyMethod;
```

This way you *force* the compiler to start searching from an ancestor class. In our example, it means that compiler is searching for `MyMethod` inside `TMyClass1.MyMethod`, then `TObject.MyMethod`, and then gives up. It does not even consider using the implementation of `TMyClass2.MyMethod`.

> Go ahead, change the implementation of `TMyClass2.MyOtherMethod` above to use `inherited MyMethod`, and see the difference in the output.

The `inherited` call is often used to call the ancestor method of the same name. This way the descendants can enhance the ancestors (keeping the ancestor functionality, instead of replacing the ancestor functionality). Like in the example below.

```
{$mode objfpc}{$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.
```

Since using `inherited` to call a method with the same name, with the same arguments, is a very often case, there is a special shortcut for it: you can just write `inherited;` (`inherited` keyword followed immediately by a semicolon, instead of a method name). This means "*call an inherited method with the same name, passing it the same arguments as the current method*".

In the above example, all the `inherited …;` calls could be replaced by a simple `inherited;`.

Note 1: The `inherited;` is really just a shortcut for calling the ancestor's method with the *same variables passed in*. If you have modified your own parameter (which is possible, if the parameter is not `const`), then the ancestor's method can receive different input values from your descendant. Consider this:

```
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;
```

Note 2: You usually want to make the `MyMethod` *virtual* when many classes (along the "*inheritance chain*") define it. More about the virtual methods in the section below. But the `inherited` keyword works regardless if the method is virtual or not. The `inherited` always means that the compiler starts searching for the method in an ancestor, and it makes sense for both *virtual* and *not virtual* methods.

4.9. Virtual methods, override and reintroduce

By default, the methods are *not virtual*. This is similar to C++, and unlike Java.

When a method is *not virtual*, the compiler determines which method to call based on the currently *declared* class type, not based on the *actually created* class type. The difference seems subtle, but it's important when your variable is declared to have a class like `TFruit`, but it may be in fact a descendant class like `TApple`.

The idea of the object-oriented programming is that *the descendant class is always as good as the ancestor*, so the compiler allows to use a descendant class always when the ancestor is expected. When your method is not virtual, this can have undesired consequences. Consider the example below:

```
{$mode objfpc}{$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.
```

This example will print

```
We have a fruit with class TApple
We eat it:
Eating a fruit
```

In effect, the call `Fruit.Eat` called the `TFruit.Eat` implementation, and nothing calls the `TApple.Eat` implementation.

If you think about how the compiler works, this is natural: when you wrote the `Fruit.Eat`, the `Fruit` variable was declared to hold a class `TFruit`. So the

compiler was searching for the method called `Eat` within the `TFruit` class. If the `TFruit` class would not contain such method, the compiler would search within an ancestor (`TObject` in this case). But the compiler *cannot search within descendants (like `TApple`,)* as it doesn't know whether the *actual class* of `Fruit` is `TApple`, `TFruit`, or some other `TFruit` descendant (like a `TOrange`, not shown in the example above).

In other words, the *method to be called* is determined *at compile-time*.

Using the *virtual methods* changes this behavior. **If the `Eat` method would be virtual** (an example of it is shown below), then the actual implementation to be called is determined *at runtime*. If the `Fruit` variable will hold an instance of the class `TApple` (even if it's declared as `TFruit`), then the `Eat` method will be searched within the `TApple` class first.

In Object Pascal, to define a method as *virtual*, you need to

- Mark it's first definition (in the top-most ancestor) with the `virtual` keyword.
- Mark all the other definitions (in the descendants) with the `override` keyword. All the overridden versions must have exactly the same parameters (and return the same types, in case of functions).

```
{$mode objfpc}{$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.
```

This example will print

```
We have a fruit with class TApple
We eat it:
Eating an apple
```

Internally, virtual methods work by having so-called *virtual method table* associated with each class. This table is a list of pointers to the implementations of virtual methods for this class. When calling the `Eat` method, the compiler looks into a virtual method table associated with the actual class of `Fruit`, and uses a pointer to the `Eat` implementation stored there.

If you don't use the `override` keyword, the compiler will warn you that you're *hiding* (obscuring) the virtual method of an ancestor with a non-virtual definition. If you're sure that this is what you want, you can add a `reintroduce` keyword. But in most cases, you will rather want to keep the method virtual, and add the `override` keyword, thus making sure that it's always invoked correctly.

5. Freeing classes

5.1. Remember to free the class instances

The class instances have to be manually freed, otherwise you get memory leaks. I advice using FPC `-gl -gh` options to detect memory leaks (see https://castle-engine.sourceforge.io/manual_optimization.php#section_memory).

Note that this doesn't concern raised exceptions. Although you do create a class when raising an exception (and it's a perfectly normal class, and you can create your own classes for this purpose too). But this class instance is freed automatically.

5.2. How to free

To free the class instance, it's best to call `FreeAndNil(A)` on your class instance. It checks whether `A` is `nil`, if not — calls it's destructor, and sets `A` to `nil`. So calling it many times in a row is not an error.

It is more-or-less a shortcut for

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

Actually, that's an oversimplification, as `FreeAndNil` does a useful trick and sets the variable `A` to `nil` **before** calling the destructor on a suitable reference. This helps to prevent a certain class of bugs — the idea is that the "outside" code should never access a half-destructed instance of the class.

Often you will also see people using the `A.Free` method, which is like doing

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

This frees the `A`, unless it's `nil`.

Note that in normal circumstances, you should never call a method on an instance which may be `nil`. So the call `A.Free` may look suspicious at the first sight, if `A` can be `nil`. However, the `Free` method is an exception to this rule. It does something dirty in the implementation — namely, checks whether `Self <> nil`. This dirty trick works only in non-virtual methods (that don't call any virtual methods and don't access any fields).

I advice using `FreeAndNil(A)` always, without exceptions, and never to call directly the `Free` method or `Destroy` destructor. The *Castle Game Engine* does it like that. It helps to keep a nice assertion that *all references are either nil, or point to valid instances*.

5.3. Manual and automatic freeing

In many situations, the need to free the instance is not much problem. You just write a destructor, that matches a constructor, and deallocates everything that was allocated in the constructor (or, more completely, in the whole lifetime of the class). Be careful to only free each thing **once**. Usually it's a good idea to set the freed reference to `nil`, usually it's most comfortable to do it by calling the `FreeAndNil(A)`.

So, like this:

```
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;
```

To avoid the need to explicitly free the instance, one can also use the `TComponent` feature of *"ownership"*. An object that is *owned* will be automatically freed by the *owner*. The mechanism is smart and it will never free an already freed instance (so things will also work correctly if you manually free the owned object earlier). We can change the previous example to this:

```
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;
```

Note that we need to override a virtual `TComponent` constructor here. So we cannot change the constructor parameters. (Actually, you can — declare a new constructor with `reintroduce`. But be careful, as some functionality, e.g. streaming, will still use the virtual constructor, so make sure it works right in either case.)

Another mechanism for automatic freeing is the `OwnsObjects` functionality (by default already `true`!) of list-classes like `TFPGObjectList` or `TObjectList`. So we can also write:

```
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;
```

Beware that the list classes "ownership" mechanism is simple, and you will get an error if you free the instance using some other means, while it’s also contained within a list. Use the `Extract` method to remove something from a list without freeing it, thus taking the responsibility to free it yourself.

In the Castle Game Engine: The descendants of `TX3DNode` have automatic memory management when inserted as children of another `TX3DNode`. The root X3D node, `TX3DRootNode`, is in turn usually owned by `TCastleSceneCore`. Some other things also have a simple ownership mechanism — look for parameters and properties called `OwnsXxx`.

5.4. The virtual destructor called Destroy

As you saw in the examples above, when the class is destroyed, it's `destructor` called `Destroy` is called.

In theory, you could have multiple destructors, but in practice it's almost never a good idea. It's much easier to have only one destructor called `Destroy`, which is in turn called by the `Free` method, which is in turn called by the `FreeAndNil` procedure.

The `Destroy` destructor in the `TObject` is defined as a *virtual* method, so you should always mark it with the `override` keyword in all your classes (since all classes descend from `TObject`). This makes the `Free` method work correctly. Recall how the virtual methods work from the Section 4.9, "Virtual methods, override and reintroduce".

This information about *destructors* is, indeed, inconsistent with the *constructors*.

It's normal that a class has multiple constructors. Usually they are all called `Create`, and only have different parameters, but it's also OK to invent other names for constructors.

Also, the `Create` constructor in the `TObject` is *not virtual*, so you do not mark it with `override` in the descendants.

This all gives you a bit of extra flexibility when defining constructors. It is often not necessary to make them virtual, so by default you're not forced to do it.

Note, however, that this changes for `TComponent` descendants. The `TComponent` defines a virtual constructor `Create(AOwner: TComponent)`. It needs a virtual constructor in order for the streaming system to work. When defining descendants of the `TComponent`, you should override this constructor (and mark it with the `override` keyword), and perform all your initialization inside it. It is still OK to define additional constructors, but they should only act as *"helpers"*. The instance should always work when created using the `Create(AOwner: TComponent)` constructor, otherwise it will not be correctly constructed when streaming. The *streaming* is used e.g. when saving and loading this component on a Lazarus form.

5.5. Free notification

If you copy a reference to the instance, such that you have two references to the same memory, and then one of them is freed — the other one becomes a *"dangling pointer"*. It should not be accessed, as it points to a memory that is no longer allocated. Accessing it may result in a runtime error, or garbage being returned (as the memory may be reused for other stuff in your program).

Using the `FreeAndNil` to free the instance doesn't help here. `FreeAndNil` sets to `nil` only the reference it got — there's no way for it to set all other references. Consider this code:

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

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

1. At the end of this block, the `O1` is `nil`. If some code has to access it, it can reliably use `if O1 <> nil …` to avoid calling methods on a freed instance, like

   ```
   if O1 <> nil then
     Writeln(O1.ClassName);
   ```

   Trying to access a field of a `nil` instance results in a predictable exception at runtime. So even if some code will not check `O1 <> nil`, and will blindly access `O1` field, you will get a clear exception at runtime.

   Same goes for calling a virtual method, or calling a non-virtual method that accessed a field of a `nil` instance.

2. With `O2`, things are less predictable. It's not `nil`, but it's invalid. Trying to access a field of a non-nil invalid instance results in an unpredictable behavior — maybe an access violation exception, maybe a garbage data returned.

There are various solutions to it:

- One solution is to, well, be careful and read the documentation. Don't assume anything about the lifetime of the reference, if it's created by other code. If a class `TCar` has a field pointing to some instance of `TWheel`, it's a *convention* that the reference to *wheel* is valid as long as the reference to *car* exists, and the *car* will free it's *wheels* inside it's destructor. But that's just a convention, the documentation should mention if there's something more complicated going on.
- In the above example, right after freeing the `O1` instance, you can simply set the `O2` variable explicitly to `nil`. That's trivial in this simple case.
- The most future-proof solution is to use `TComponent` class "free notification" mechanism. One component can be notified when another component is freed, and thus set it's reference to `nil`.

  Thus you get something like a *weak reference*. It can cope with various usage scenarios, for example you can let the code from outside of the class to set your reference, and the outside code can also free the instance at anytime.

  This requires both classes to descend from `TComponent`. Using it in general boils down to calling `FreeNotification` , `RemoveFreeNotification`, and overriding `Notification`.

  Here's a complete example, showing how to use this mechanism, together with constructor / destructor and a setter property. Sometimes it can be done simpler, but this is the full-blown version that is always correct:)

```
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;
```

6. Run-time library

6.1. Input/output using streams

Modern programs should use `TStream` class and it's many descendants to do input / output. It has many useful descendants, like `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.
```

In the Castle Game Engine: You should use the `Download` method to create a stream that operates of resources (which includes files, data downloaded from URLs and Android assets). Moreover, to open the resource inside your game data (typically in `data` subdirectory) use the `ApplicationData` function.

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

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

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

To read text files, we advice using the `TTextReader` class. It provides a line-oriented API, and wraps a `TStream` inside. The `TTextReader` constructor can take a ready URL, or you can pass there your custom `TStream` source.

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

6.2. Containers (lists, dictionaries) using generics

The language and run-time library offer various flexible containers. There are a number of non-generic classes (like `TList` and `TObjectList` from the `Contnrs` unit),

there are also dynamic arrays (`array of TMyType`). But to get the most flexibility **and** type-safety, I advice using **generic containers** for most of your needs.

The *generic containers* give you a lot of helpful methods to add, remove, iterate, search, sort… The compiler also knows (and checks) that the container holds only items of the appropriate type.

There are three libraries providing generics containers in FPC now:

- `FGL` unit
- `Generics.Collections` unit and friends
- `GVector` unit and friends (together in `fcl-stl`)

We advice using the `FGL` unit (if you need to work with the latest stable FPC 3.0.0 and 3.0.2 or even older FPC 2.6.x), or the `Generics.Collections` unit (only available since FPC 3.1.1, but on the other hand compatible with Delphi). They both offer lists and dictionaries with naming consistent with other parts of the standard library (like the non-generic containers from the `Contnrs` unit).

In the Castle Game Engine: We include a local copy of `Generics.Collections` even for FPC 3.0.0 and 3.0.2. We use the `Generics.Collections` throughout the engine, and advice you to use `Generics.Collections` too!

Most important classes from the `Generics.Collections` unit are:

TList
: A generic list of types.

TObjectList
: A generic list of object instances. It can "own" children, which means that it will free them automatically.

TDictionary
: A generic dictionary.

TObjectDictionary
: A generic dictionary, that can "own" the keys and/or values.

Here's how to you use a simple generic `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.
```

Note that some operations require comparing two items, like sorting and searching (e.g. by `Sort` and `IndexOf` methods). The `Generics.Collections` containers use for this a *comparer*. The *default comparer* is reasonable for all types, even for records (in which case it compares memory contents, which is a reasonable default at least for searching using `IndexOf`).

When sorting the list you can provide a *custom comparer* as a parameter. The *comparer* is a class implementing the `IComparer` interface. In practice, you usually define the appropriate callback, and use `TComparer<T>.Construct` method to wrap this callback into an `IComparer` instance. An example of doing this is below:

```
{$mode objfpc}{$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.
```

The `TDictionary` class implements a **dictionary**, also known as a **map (key → value)**, also known as an **associative array**. It's API is a bit similar to the C# `TDictionary` class. It has useful iterators for keys, values, and pairs of key→value.

An example code using a dictionary:

```
{$mode objfpc}{$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.
```

The `TObjectDictionary` can additionally *own* the dictionary keys and/or values, which means that they will be automatically freed. Be careful to *only own keys and/or values if they are object instances*. If you set to *"owned"* some other type, like an `Integer` (for example, if your keys are `Integer`, and you include `doOwnsKeys`), you will get a nasty crash when the code executes.

An example code using the `TObjectDictionary` is below. Compile this example with *memory leak detection*, like `fpc -gl -gh generics_object_dictionary.lpr`, to see that everything is freed when program exits.

```
{$mode objfpc}{$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.
```

If you prefer using the `FGL` unit instead of `Generics.Collections`, here is a short overview of the most important classes from the `FGL` unit:

TFPGList
: A generic list of types.

TFPGObjectList
: A generic list of object instances. It can "own" children.

TFPGMap
: A generic dictionary.

In `FGL` unit, the `TFPGList` can be only used for types for which the equality operator (=) is defined. For `TFPGMap` the *"greater than"* (>) and *"less than"* (<) operators must be defined for the key type. If you want to use these lists with types that don't have built-in comparison operators (e.g. with records), you have to overload their operators as shown in the Section 7.9, "Operator overloading".

In the Castle Game Engine we include a unit `CastleGenericLists` that adds `TGenericStructList` and `TGenericStructMap` classes. They are similar to `TFPGList` and `TFPGMap`, but they do not require a definition of the comparison operators for the appropriate type (instead, they compare memory contents, which is often appropriate for records or method pointers). But the `CastleGenericLists` unit is deprecated since the engine version 6.3, as we advice using `Generics.Collections` instead.

If you want to know more about the generics, see Section 7.3, "Generics".

6.3. Cloning: TPersistent.Assign

Copying the class instances by a simple assignment operator copies the **reference**.

```
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;
```

To copy the **class instance contents**, the standard approach is to derive your class from `TPersistent`, and override it's `Assign` method. Once it's implemented properly in `TMyObject`, you use it like this:

```
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;
```

To make it work, you need to implement the `Assign` method to actually copy the fields you want. You should carefully implement the `Assign` method, to copy from a class that may be a descendant of the current class.

```
{$mode objfpc}{$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);
    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.
```

Sometimes it's more comfortable to alternatively override the `AssignTo` method in the source class, instead of overriding the `Assign` method in the destination class.

Be careful when you call `inherited` in the overridden `Assign` implementation. There are two situations:

Your class is a direct descendant of the `TPersistent` class. (Or, it's not a direct descendant of `TPersistent`, but no ancestor has overridden the `Assign` method.)

In this case, your class should use the `inherited` keyword (to call the `TPersistent.Assign`) *only if you cannot handle the assignment in your code*.

Your class descends from some class that has already overridden the `Assign` method.

In this case, your class should *always* use the `inherited` keyword (to call the ancestor `Assign`). In general, calling `inherited` in overridden methods is *usually* a good idea.

To understand the need when to call (or not to call) `inherited` from the `Assign` implementation, and how it relates to the `AssignTo` method, it's best to look at the `TPersistent.Assign` and `TPersistent.AssignTo` implementations:

```
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;
```

This is not the **exact** implementation of `TPersistent`, I simplified it to hide the boring details about how the exception message is build.

The conclusions you can get from the above are:

- *If neither `Assign` nor `AssignTo` are overridden*, then calling them will result in an exception.
- Also, note that there is *no* code in `TPersistent` implementation that automatically copies all the fields (or all the published fields) of the classes. That's why you need to do that yourself, by overriding `Assign` in all the classes. You can use RTTI (runtime type information) for that, but for simple cases you will probably just list the fields to be copied manually.

When you have a class like `TApple`, your `TApple.Assign` implementation usually deals with copying fields that are specific to the `TApple` class (not to the `TApple` ancestor, like `TFruit`). So, the `TApple.Assign` implementation usually checks whether `Source is TApple` at the beginning, before copying apple-related fields. Then, it calls `inherited` to allow `TFruit` to handle the rest of the fields.

Assuming that you implemented `TFruit.Assign` and `TApple.Assign` following the standard pattern (as shown in the example above), the effect is like this:

- If you pass `TApple` instance to `TApple.Assign`, it will work and copy all the fields.
- If you pass `TOrange` instance to `TApple.Assign`, it will work and only copy the common fields shared by both `TOrange` and `TApple`. In other words, the fields defined at `TFruit`.
- If you pass `TWerewolf` instance to `TApple.Assign`, it will raise an exception (because `TApple.Assign` will call `TFruit.Assign` which will call `TPersistent.Assign` which raises an exception).

Remember that when descending from `TPersistent`, the default *visibility specifier* is `published`, to allow streaming of `TPersistent` descendants. Not all field and property types are allowed in the `published` section. If you get errors related to it, and

> you don't care about streaming, just change the visibility to `public`. See the Section 4.5, "Visibility specifiers".

7. Various language features

7.1. Local (nested) routines

Inside a larger *routine* (function, procedure, method) you can define a helper routine.

The local routine can freely access (read and write) all the parameters of a parent, *and all the local variables of the parent that were declared above it*. This is very powerful. It often allows to split long routines into a couple of small ones without much effort (as you don't have to pass around all the necessary information in the parameters). Be careful to not overuse this feature — if many nested functions use (and even change) the same variable of the parent, the code may get hard to follow.

These two examples are equivalent:

```
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;
```

Another version, where we let the local routine `Square` to access `I` directly:

```
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;
```

Local routines can go to any depth — which means that you can define a local routine within another local routine. So you can go wild (but please don't go *too wild*, or the code will get unreadable:).

7.2. Callbacks (aka events, aka pointers to functions, aka procedural variables)

They allow to call a function indirectly, through to a variable. The variable can be assigned at runtime to point to any function *with matching parameter types and return types*.

The callback can be:

- Normal, which means it can point to any normal routine (not a method, not local).

```
{$mode objfpc}{$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.
```

- A method: declare with `of object` at the end.

```
{$mode objfpc}{$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.
```

Note that you *cannot* pass global procedures / functions as methods. They are incompatible. If you have to provide an `of object` callback, but don't want to create a dummy class instance, you can pass Section 8.3, "Class methods" as methods.

```
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;
```

Unfortunately, you need to write ugly `@TMyClass(nil).Add` instead of just `@TMyClass.Add`.

- A (possibly) local routine: declare with `is nested` at the end, and make sure to use `{$modeswitch nestedprocvars}` directive for the code. These go hand-in-hand with Section 7.1, “Local (nested) routines”.

7.3. Generics

A powerful feature of any modern language. The definition of something (typically, of a class) can be parameterized with another type. The most typical example is when you need to create a container (a list, dictionary, tree, graph…): you can define *a list of type T*, and then *specialize* it to instantly get *a list of integers*, *a list of strings*, *a list of TMyRecord*, and so on.

The generics in Pascal are realized much like generics in C++. Which means that they are *"expanded"* at the specialization time, a *little* like macros (but much safer than macros; for example, the identifiers are resolved at the time of generic definition, not at specialization, so you cannot "inject" any unexpected behavior when specializing the generic). In effect this means that they are very fast (can be optimized for each particular type) and work with types of any size. You can use a primitive type (integer, float) as well as a record as well as a class when specializing a generic.

```
{$mode objfpc}{$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.
```

Generics are not limited to classes, you can have generic functions and procedures as well:

```
{$mode objfpc}{$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.
```

See also the Section 6.2, “Containers (lists, dictionaries) using generics” about important standard classes using generics.

7.4. Overloading

Methods (and global functions and procedures) with the same name are allowed, as long as they have different parameters. At compile time, the compiler detects which one you want to use, knowing the parameters you pass.

By default, the overloading uses the FPC approach, which means that all the methods in given namespace (a class or a unit) are equal, and hide the other methods in namespaces with less priority. For example, if you define a class with methods `Foo(Integer)` and `Foo(string)`, and it descends from a class with method `Foo(Float)`, then the users of your new class will not be able to access the method `Foo(Float)` easily (they still can --- if they typecast the class to it's ancestor type). To overcome this, use the `overload` keyword.

7.5. Preprocessor

You can use simple preprocessor directives for

- conditional compilation (code depending on platform, or some custom switches),
- to include one file in another,
- you can also use parameter-less macros.

Note that macros with parameters are not allowed. In general, you should avoid using the preprocessor stuff… unless it's really justified. The preprocessing happens before parsing, which means that you can "break" the normal syntax of the Pascal language. This is a powerful, but also somewhat dirty, feature.

```
{$mode objfpc}{$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 files have commonly the `.inc` extension, and are used for two purposes:

- The include file may only contain other compiler directives, that "configure" your source code. For example you could create a file `myconfig.inc` with these contents:

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

  Now you can include this file using `{$I myconfig.inc}` in all your sources.

- The other common use is to split a large unit into many files, while still keeping it a single unit as far as the language rules are concerned. Do not overuse this technique — your first instinct should be to split a single unit into multiple units, not to split a single unit into multiple include files. Never the less, this is a useful technique.

  1. It allows to avoid "exploding" the number of units, while still keeping your source code files short. For example, it may be better to have a single unit with *"commonly used UI controls"* than to create *one unit for each UI control class*, as the latter approach would make the typical "uses" clause long (since a typical UI code will depend on a couple of UI classes). But placing all these UI classes in a single `myunit.pas` file would make it a long file, unhandy to navigate, so splitting it into multiple include files may make sense.
  2. It allows to have a cross-platform unit interface with platform-dependent implementation easily. Basically you can do

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

     Sometimes this is better than writing a long code with many `{$ifdef UNIX}`, `{$ifdef MSWINDOWS}` intermixed with normal code (variable declarations, routine implementation). The code is more readable this way. You can even use this technique more aggressively, by using the `-Fi` command-line option of FPC to include some subdirectories only for

specific platforms. Then you can have many version of include file `{$I my platform_specific_implementation.inc}` and you simply include them, letting the compiler find the correct version.

7.6. Records

Record is just a container for other variables. It's like a much, much simplified *class*: there is no inheritance or virtual methods. It is like a *structure* in C-like languages.

If you use the `{$modeswitch advancedrecords}` directive, records **can** have methods and visibility specifiers. In general, language features that are available for classes, and *do not break the simple predictable memory layout of a record*, are then possible.

```
{$mode objfpc}{$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.WritelnDescription;
end.
```

In modern Object Pascal, your first instinct should be to design a `class`, not a `record` — because classes are packed with useful features, like constructors and inheritance.

But records are still very useful when you need speed or a predictable memory layout:

- Records do not have any constructor or destructor. You just define a variable of a record type. It has undefined contents (memory garbage) at the beginning (except auto-managed types, like strings; they are guaranteed to be initialized to be empty, and finalized to free the reference count). So you have to be more careful when dealing with records, but it gives you some performance gain.
- Arrays of records are nicely linear in memory, so they are cache-friendly.
- The memory layout of records (size, padding between fields) is clearly defined in some situations: when you request the *C layout*, or when you use `packed record`. This is useful:
  - to communicate with libraries written in other programming languages, when they expose an API based on records,
  - to read and write binary files,
  - to make dirty low-level tricks (like unsafe typecasting one type to another, being aware of their memory representation).
- Records can also have `case` parts, which work like *unions* in C-like languages. They allows to treat the same memory piece as a different type, depending on your needs. As such, this allows for greater memory efficiency in some cases. And it allows for more *dirty, low-level unsafe tricks*:)

7.7. Old-style objects

In the old days, Turbo Pascal introduced another syntax for class-like functionality, using the `object` keyword. It's somewhat of a blend between the concept of a `record` and a modern `class`.

- The old-style objects can be allocated / freed, and during that operation you can call their constructor / destructor.
- But they can also be simply declared and used, like records. A simple `record` or `object` type is not a reference (pointer) to something, it's simply the data. This makes them comfortable for small data, where calling allocation / free would be bothersome.

- Old-style objects offer inheritance and virtual methods, although with small differences from the modern classes. Be careful — *bad things* will happen if you try to use an object without calling it's constructor, and the object has virtual methods.

It's discouraged to use the old-style objects in most cases. Modern *classes* provide much more functionality. And when needed, records (including *advanced records*) can be used for performance. These concepts are usually a better idea than old-style objects.

7.8. Pointers

You can create a *pointer* to any other type. The pointer to type `TMyRecord` is declared as `^TMyRecord`, and by convention is called `PMyRecord`. This is a traditional example of a linked list of integers using records:

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

Note that the definition is recursive (type `PMyRecord` is defined using type `TMyRecord`, while `TMyRecord` is defined using `PMyRecord`). It is allowed to define a pointer type to a *not-yet-defined type*, as long as it will be resolved within the same `type` block.

You can allocate and free pointers using the `New` / `Dispose` methods, or (more low-level, not type-safe) `GetMem` / `FreeMem` methods. You dereference the pointer (to access the stuff *pointed by*) you append the `^` operator. To make the inverse operation, which is to *get a pointer of an existing variable*, you prefix it with `@` operator.

There is also an untyped `Pointer` type, similar to `void*` in C-like languages. It is completely unsafe, and can be typecasted to any other pointer type.

Remember that a *class instance* is also in fact a pointer, although it doesn't require any `^` or `@` operators to use it. A linked list using classes is certainly possible, it would simply be this:

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

```
    Next: TMyClass;
  end;
```

7.9. Operator overloading

You can override the meaning of many language operators, for example to allow addition and multiplication of your custom types. Like this:

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

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

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

You can override operators on classes too. Since you usually create new instances of your classes inside the operator function, the caller must remember to free the result.

```
{$mode objfpc}{$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.
```

You can override operators on records too. This is usually easier than overloading them for classes, as the caller doesn't have to deal then with memory management.

```
{$mode objfpc}{$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.
```

For records, it's adviced to use `{$modeswitch advancedrecords}` and override operators as `class operator` inside the record. This allows to use generic classes that depend on some operator existence (like `TFPGList`, that depends on equality operator being available) with such records. Otherwise the "global" definition of an operator (not inside the record) would not be found (because it's not available at the code that implements the `TFPGList`), and you could not specialize a list like `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. Advanced classes features

8.1. private and strict private

The `private` visibility specifier means that the field (or method) is not accessible outside of this class. But it allows an exception: all the code defined *in the same unit* can

break this, and access private fields and methods. A C++ programmer would say that in Pascal *all classes within a single unit are friends*. This is often useful, and doesn't break your encapsulation, since it's limited to a unit.

However, if you create larger units, with many classes (that are not tightly integrated with each other), it's safer to use `strict private`. As you can guess, it means that the field (or method) in not accessible outside of this class — period. No exceptions.

In a similar manner, there's `protected` visibility (visible to descendants, or friends in the same unit) and `strict protected` (visible to descendants, period).

8.2. More stuff inside classes and nested classes

You can open a section of constants (`const`) or types (`type`) within a class. This way, you can even define a class within a class. The visibility specifiers work as always, in particular the nested class can be private (not visible to the outside world), which is often useful.

Note that to declare a field after a constant or type you will need to open a `var` block.

```
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. Class methods

These are methods you can call having a class reference (`TMyClass`), not necessarily a class instance.

```
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;
```

Note that they can be virtual — it makes sense, and is sometimes very useful, when combined with Section 8.4, “Class references”.

The class methods can also be limited by the Section 4.5, “Visibility specifiers”, like `private` or `protected` . Just like regular methods.

Note that a constructor always acts like a class method when called in a normal fashion (`MyInstance := TMyClass.Create(…);`). Although it’s possible to also call a constructor from within the class itself, like a normal method, and then it acts like a

normal method. This is a useful feature to "chain" constructors, when one constructor (e.g. overloaded to take an integer parameter) does some job, and then calls another constructor (e.g. parameter-less).

8.4. Class references

Class reference allows you to choose the class at runtime, for example to call a class method or constructor without knowing the exact class at compile-time. It is a type declared as `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);

  // 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;
```

Class references can be combined with virtual class methods. This gives a similar effect as using classes with virtual methods — the actual method to be executed is determined at runtime.

```
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;

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

If you have an instance, and you would like to get a reference to it's class (not the declared class, but the final descendant class used at its construction), you can use the `ClassType` property. The declared type of `ClassType` is `TClass`, which stands for `class of TObject`. Often you can safely typecast it to something more specific, when you know that the instance is something more specific than `TObject`.

In particular, you can use the `ClassType` reference to call virtual methods, including virtual constructors. This allows you to create a method like `Clone` that constructs

an instance of *the exact run-time class of the current object*. You can combine it with Section 6.3, “Cloning: TPersistent.Assign” to have a method that returns a newly-constructed clone of the current instance.

Remember that it only works when the constructor of your class is virtual. For example, it can be used with the standard `TComponent` descendants, since they all must override `TComponent.Create(AOwner: TComponent)` virtual constructor.

```
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. Static class methods

To understand the *static class methods*, you have to understand how the *normal class methods* (described in the previous sections) work. Internally, *normal class methods* receive a *class reference* of their class (it is passed through a hidden, implicitly added 1st parameter of the method). This class reference can be even accessed explicitly using the `Self` keyword inside the class method. Usually, it’s a good thing: this class reference allows you to call *virtual class methods* (through the *virtual method table* of the class).

While this is nice, it makes the *normal class methods* incompatible when assigning to a *global procedure pointer*. That is, **this will not compile**:

```
{$mode objfpc}{$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.
```

In the *Delphi mode* you would be able to write `TMyClass.Foo` instead of an ugly `TMyClass(nil).Foo` in the example above. Admittedly, the `TMyClass.Foo` looks much more elegant, and it is also better checked by the compiler. Using the `TMyClass(nil).Foo` is a hack… unfortunately, necessary (for now) in the *ObjFpc mode* which is presented throughout this book.

In any case, assigning the `TMyClass.Foo` to the `Callback` above would *still fail* in the Delphi mode, for exactly the same reasons.

The above example fails to compile, because the `Callback` is incompatible with the class method `Foo`. And it's incompatible because internally the class method has that special hidden *implicit* parameter to pass a class reference.

One way to fix the above example is to change the definition of `TMyCallback`. It will work if it is a method callback, declared as `TMyCallback = procedure (A: Integer) of object;`. But sometimes, it's not desirable.

Here comes the `static` class method. It is, in essence, just a global procedure / function, but it's namespace is limited inside the class. It *does not* have any implicit class reference (and so, *it cannot be virtual* and *it cannot call virtual class methods*). On the upside, it is compatible with normal (non-object) callbacks. So this will work:

```
{$mode objfpc}{$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. Class properties and variables

A *class property* is a property that can be accessed through a class reference (it does not need a class instance).

It is quite straightforward analogy of a regular property (see Section 4.3, “Properties”). For a *class property*, you define a *getter* and / or a *setter*. They may refer to a *class variable* or a *static class method*.

A *class variable* is, you guessed it, like a regular field but you don’t need a class instance to access it. In effect, it’s just like a global variable, but with the namespace limited to the containing class. It can be declared within the `class var` section of the class. Alternatively it can be declared by following the normal field definition with the keyword `static`.

And a *static class method* is just like a global procedure / function, but with the namespace limited to the containing class. More about static class methods in the section above, see Section 8.5, “Static class methods”.

```
{$mode objfpc}{$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. Class helpers

The *method* is just a procedure or function inside a class. From the outside of the class, you call it with a special syntax `MyInstance.MyMethod(…)`. After a while you grow accustomed to thinking that *if I want to make action Action on instance X, I write `X.Action(…)`*.

But sometimes, you need to implement something that conceptually is *an action on class TMyClass* without modifying the *TMyClass* source code. Sometimes it's because it's not your source code, and you don't want to change it. Sometimes it's because of the dependencies — adding a method like `Render` to a class like `TMy3DObject` seems like a straightforward idea, but maybe the base implementation of class `TMy3DObject` should be kept independent from the rendering code? It would be better to "enhance" an existing class, to add functionality to it without changing it's source code.

Simple way to do it is then to create a global procedure that takes an instance of `TMy3DObject` as it's 1st parameter.

```
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;
```

This works perfectly, but the downside is that calling it looks a little ugly. While usually you call actions like `X.Action(…)`, in this case you have to call them like `Render(X, …)`. It would be cool to be able to just write `X.Render(…)`, even when `Render` is not implemented in the same unit as `TMy3DObject`.

And this is where you use class helpers. They are just a way to implement procedures / functions that operate on given class, and that are called like methods, but are not in fact normal methods — they were added outside of the `TMy3DObject` definition.

```
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;
```

The more general concept is *"type helper"*. Using them you can add methods even to primitive types, like integers or enums. You can also add *"record helpers"* to (you guessed it…) records. See http://lists.freepascal.org/fpc-announce/2013-February/000587.html .

8.8. Virtual constructors, destructors

Destructor name is always `Destroy`, it is virtual (since you can call it without knowing the exact class at compile-time) and parameter-less.

Constructor name is by convention `Create`.

You can change this name, although be careful with this — if you define `CreateMy`, always redefine also the name `Create`, otherwise the user can still access the constructor `Create` of the ancestor, bypassing your `CreateMy` constructor.

In the base `TObject` it is not virtual, and when creating descendants you're free to change the parameters. The new constructor will hide the constructor in ancestor (note: don't put here `overload`, unless you want to break it).

In the `TComponent` descendants, you should override it's `constructor Create(AOwner: TComponent);`. For streaming functionality, to create a class without knowing it's type at compile time, having virtual constructors is very useful (see "class references" below).

8.9. An exception in constructor

What happens if an exception happens during a constructor? The line

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

does not execute to the end in this case, `X` cannot be assigned, so who will cleanup after a partially-constructed class?

The solution of Object Pascal is that, in case an exception occurs within a constructor, then the destructor is called. This is a reason why *your destructor must be robust*, which means it should work in any circumstances, even on a half-created class instance. Usually this is easy if you release everything safely, like by `FreeAndNil`.

We also have to depend in such cases that *the memory of the class is guaranteed to be zeroed right before the constructor code is executed*. So we know that at the beginning, all class references are `nil`, all integers are `0` and so on.

So below works without any memory leaks:

```
{$mode objfpc}{$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.
```

9. Interfaces

9.1. Bare (CORBA) interfaces

An interface declares an API, much like a class, but it does not define the implementation. A class can implement many interfaces, but it can only have one ancestor class.

You can cast a class to any interface it supports, and then *call the methods through that interface*. This allows to treat in a uniform fashion the classes that don't descend from each other, but still share some common functionality. Useful when a simple class inheritance is not enough.

The *CORBA interfaces* in Object Pascal work pretty much like interfaces in Java (https://docs.oracle.com/javase/tutorial/java/concepts/interface.html) or C# (https://msdn.microsoft.com/en-us/library/ms173156.aspx).

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

```
uses SysUtils, Classes;

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

  TMyClass1 = class(IMyInterface)
    procedure Shoot;
  end;

  TMyClass2 = class(IMyInterface)
    procedure Shoot;
  end;

  TMyClass3 = class
    procedure Shoot;
  end;

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

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

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

procedure UseThroughInterface(I: IMyInterface);
begin
  Write('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 and COM types of interfaces

Why are the interfaces (presented above) called "CORBA"?

The name **CORBA** is unfortunate. A better name would be **bare interfaces**. These interfaces are a *"pure language feature"*. Use them when you want to cast various classes as the same interface, because they share a common API.

While these types of interfaces can be used together with the *CORBA (Common Object Request Broker Architecture) technology* (see https://en.wikipedia.org/wiki/Common_Object_Request_Broker_Architecture), they are *not* tied to this technology in any way.

Is the `{$interfaces corba}` declaration needed?

Yes, because by default you create *COM interfaces*. This can be stated explicitly by saying `{$interfaces com}`, but usually it's not needed since it's the default state.

And I don't advice using *COM interfaces*, especially if you're looking for something equivalent to interfaces from other programming languages. The *CORBA interfaces* in Pascal are exactly what you expect if you're looking for something equivalent to the interfaces in C# and Java. While the *COM interfaces* bring additional features that you possibly don't want.

Note that the `{$interfaces xxx}` declaration only affects the interfaces that do not have any explicit ancestor (just the keyword `interface`, not `interface(ISomeAncestor)`). When an interface has an ancestor, it has the same type as the ancestor, regardless of the `{$interfaces xxx}` declaration.

What are COM interfaces?

The *COM interface* is synonymous with *an interface descending from a special `IUnknown` interface*. Descending from `IUnknown`:

- Requires that your classes define the `_AddRef` and `_ReleaseRef` methods. Proper implementation of these methods can manage the lifetime of your objects using the reference-counting.
- Adds the `QueryInterface` method.
- Allows to interact with the *COM (Component Object Model) technology*.

Why do you advice to not use the COM interfaces?

Because *COM interfaces* "entangle" two features that should be unrelated (orthogonal) in my view: *multiple inheritance* and *reference counting*. Other programming languages rightly use separate concepts for these two features.

To be clear: **reference-counting**, that provides an automatic memory management (in simple situations, i.e. without cycles), **is a very useful concept**. But **entangling this feature with interfaces (instead of making them orthogonal features) is unclean in my eyes**. It definitely doesn't match my use cases.

- Sometimes I want to cast my (otherwise unrelated) classes to a common interface.
- Sometimes I want to manage memory using the reference counting approach.
- *Maybe* some day I will want to interact with the *COM technology*.

But these are all separate, unrelated needs. Entangling them in a single language feature is counter-useful in my experience. It does cause actual problems:

- If I want the feature of *casting classes to a common interface API*, but I don't want the reference-counting mechanism (I want to manually free objects), then the COM interfaces are problematic. Even when reference-counting is disabled by a special `_AddRef` and `_ReleaseRef` implementation, you still need to be careful to never have a temporary interface reference hanging, after you have freed the class instance. More details about it in the next section.
- If I want the feature of *reference counting*, but I have no need for an interface hierarchy to represent something different than the class hierarchy, then I have

to duplicate my classes API in interfaces. Thus creating a single interface for each class. This is counter-productive. I would much rather have *smart pointers* as a separate language feature, not entangled with interfaces (and luckily, it's coming:).

That is why I advice to use *CORBA* style interfaces, and the `{$interfaces corba}` directive, in all modern code dealing with interfaces.

Only if you need both "reference counting" and "multiple inheritance" at the same time, then use COM interfaces. Also, Delphi has only COM interfaces for now, so you need to use COM interfaces if your code must be compatible with Delphi.

Can we have reference-counting with CORBA interfaces?

Yeah. Just add `_AddRef` / `_ReleaseRef` methods. There's no need to descend from the `IUnknown` interface. Although in most cases, if you want reference-counting with your interfaces, you may as well just use COM interfaces.

9.3. Interfaces GUIDs

GUIDs are the seemingly random characters `['{ABCD1234-…}']` that you see placed at every interface definition. Yes, they are just random. Unfortunately, they are necessary.

The GUIDs have no meaning if you don't plan on integrating with communication technologies like *COM* nor *CORBA*. But they are necessary, for implementation reasons. Don't be fooled by the compiler, that unfortunately allows you to declare interfaces without GUIDs.

Without the (unique) GUIDs, your interfaces will be treated equal by the `is` operator. In effect, it will return `true` if your class supports *any* of your interfaces. The magic function `Supports(ObjectInstance, IMyInterface)` behaves slightly better here, as it refuses to be compiled for interfaces without a GUID. This is true for both CORBA and COM interfaces, as of FPC 3.0.0.

So, to be on the safe side, you should always declare a GUID for your interface. You can use *Lazarus* GUID generator (`Ctrl + Shift + G` shortcut in the editor). Or you can use an online service like https://www.guidgenerator.com/ .

Or you can write your own tool for this, using the `CreateGUID` and `GUIDToString` functions in RTL. See the example below:

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

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

9.4. Reference-counted (COM) interfaces

The *COM interfaces* bring two additional features:

1. integration with COM (a technology from Windows, also available on Unix through *XPCOM*, used by Mozilla),
2. reference counting (which gives you automatic destruction when all the interface references go out of scope).

When using *COM interfaces*, you need to be aware of their *automatic destruction* mechanism and relation to COM technology.

In practice, this means that:

- Your class needs to implement a magic `_AddRef`, `_Release`, and `QueryInterface` methods. Or descend from something that already implements them. A particular implementation of these methods may actually enable or disable the *reference-counting* feature of COM interfaces (although disabling it is somewhat dangerous — see the next point).
  - The standard class `TInterfacedObject` implements these methods to *enable* the reference-counting.
  - The standard class `TComponent` implements these methods to *disable* the reference-counting. **In the Castle Game Engine** we give you additional useful ancestors `TNonRefCountedInterfacedObject` and `TNonRefCountedInterfacedPersistent` for this purpose, see https://github.com/castle-engine/castle-engine/blob/master/src/base/castleinterfaces.pas .
- You need to be careful of freeing the class, when it may be referenced by some interface variables. Because the interface is released using a virtual method

(because it *may be reference-counted, even if you hack the _AddRef method to not be reference-counted…*), you cannot free the underlying object instance as long as some interface variable may point to it. See *"7.7 Reference counting"* in the FPC manual (http://freepascal.org/docs-html/ref/refse47.html).

The safest approach to using *COM interfaces* is to

- accept the fact that they are reference-counted,
- derive the appropriate classes from `TInterfacedObject`,
- and avoid using the class instance, instead accessing the instance always through the interface, letting reference-counting manage the deallocation.

This is an example of such interface use:

```
{$mode objfpc}{$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. Using COM interfaces with reference-counting disabled

As mentioned in the previous section, your class can descend from `TComponent` (or a similar class like `TNonRefCountedInterfacedObject` and `TNonRefCountedInterfacedPersistent`) which disables reference-counting for COM interfaces. This allows you to use COM interfaces, and still free the class instance manually.

You need to be careful in this case to not free the class instance when some interface variable may refer to it. Remember that every typecast `Cx as IMyInterface` also creates a temporary interface variable, which may be present even until the end of the current procedure. For this reason, the example below uses a `UseInterfaces` procedure, and it frees the class instances *outside* of this procedure (when we can be sure that temporary interface variables are out of scope).

To avoid this mess, it's usually better to use CORBA interfaces, if you don't want reference-counting with your interfaces.

```
{$mode objfpc}{$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. Typecasting interfaces without "as" operator

This section applies to both *CORBA* and *COM* interfaces.

Casting to an interface type using the `as` operator makes a check at run-time. Consider this code:

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

It works for all `C1`, `C2`, `C3` instances in the examples in previous sections. If executed, it would make a run-time error in case of `C3`, that does not implement `IMyInterface` (but we avoid the error by checking `Cx is IMyInterface` before doing the cast).

You can instead cast the instance as an interface implicitly:

```
UseThroughInterface(Cx);
```

In this case, the typecast must be valid at compile-time. So this will compile for `C1` and `C2` (that are declared as classes that implement `IMyInterface`). But it will not compile for `C3`.

In essence, this typecast looks and works just like for regular classes. Wherever an instance of a class `TMyClass` is required, you can always use there a variable that is declared with a class of `TMyClass`, **or** `TMyClass` **descendant**The same rule applies to interfaces. No need for any explicit typecast in such situations.

An equivalent is also

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

This is also a typecast that must be valid at compile-time. Note that this syntax is inconsistent with classes typecasts. In case of classes, writing `TMyClass(C)` is an *unsafe, unchecked* typecast. In case of interfaces, writing `IMyInterface(C)` is a safe, fast (checked at compile-time) typecast.

10. About this document

The source code of this document is in AsciiDoc on https://github.com/michaliskambi/modern-pascal-introduction. Suggestions for corrections and additions, and patches and pull requests, are always very welcome:) You can reach me through GitHub or email michalis.kambi@gmail.com[1]. My homepage is https://michalis.ii.uni.wroc.pl/

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

~michalis/. This document is linked under the *Documentation* section of the *Castle Game Engine* website https://castle-engine.sourceforge.io/.

Thank you for reading!