Structures in Object Oriented Programming in Free Pascal and Lazarus

Zsolt Szakály

2021.11.05

1 Preface

In my forty years journey in programming, most of the time my language of choice was Pascal. I used many other languages, but none of them I found superior to the actually most recent Pascal versions. Pascal, I was changing from mainframe implementation to microcomputers and then in the PC world from Turbo Pascal to Delphi and by now Free Pascal under Linux, using Lazarus.
The largest dilemma I regularly faced was whether to use object oriented programming (OOP) or not, and if the answer is yes, what language tools to apply.

My short conclusion that many programmers might disagree with, is to use OOP only if it makes sense and use the OOP language tool that most suits the actual need.
It is so easy to "overuse" OOP and make objects that have far too little number of fields and methods, that are used only in one copy, etc.; none of them ideal for OOP. It is safe to use more procedural programming in many cases. The counter argument is that objects, however limited their scope is at the moment, might evolve later and OOP is more flexible in that than the more classic solutions. So, make your choice wise.
In Free Pascal there are many ways how OOP can be implemented, starting from advanced **records** to **class**es. Some people argue to use the most advanced version, **class**, but my experience is that a good mix sometimes might make the software clearer or more efficient in terms of speed, etc. One strong argument against using multiple tools is that the rules of them, although similar, still different in many aspects.

To make sure what is allowed and what is not in a given language tool, I started to make experiments and made cheat-sheets for myself to remember when a class method can be static, and similar things. Over time the cheat-sheet grew pretty large and I realized that many of the topics I collected are issues others also face regularly and ask on forums.

So, now I decided to share my cheat-sheets reformatted into a kind of tutorial. I ask the readers to give any feedback, since there are still points where I was not sure, there might be topic need to be explored more in details. So, any feedback welcome, for the time-being through the Lazarus forum, where I put it first.

Please, also feel free to share the link with others, but please do not copy the document. Not only because of copyright reasons, but more, because it is very much a work in progress and I expect to rewrite it many more times. If we let it "fork", modifications, error corrections and additions will be missing from the copy.

2 Introduction

Pascal is a great language with bad, or maybe better to say, with no marketing. The problem is that Pascal is still considered to be the same as the original version released in 1970 by Niklaus Wirth. The truth is that the Pascals of today are so much more different and include so many more features that it is almost wrong to call it still just Pascal. When C was extended to cover OOP and introduced among others Classes, it got a new name, and today C people are very sensitive if one dares to call C++ simply C. Even more, C++ shows how it develops by introducing standards from C++98 to C++20, making users to feel how modern the language is.

Pascal chose a different route. Although there are some programming standards, and some well-known Pascal version names, they did not really make it to become a language as much as C++. To me Delphi, Turbo Pascal are more commercial product names than languages even if they are also language dialects. Object Pascal is a name that can also be read some places, but honestly that again is not a language in my mind, maximum a dialect.

Also, when Pascal is adding new features in the compilers or adding, upgrading new libraries, it is simply called version numbers, most of the time with full backward compatibility. Therefore we are not talking about Pascal++20 as a standard.
So, for anyone, who just familiarizes with Pascal, my main suggestion is to forget anything heard earlier about Pascal being an old, beginners only language.

In any programming language, and Pascal is no exception, if one starts to write a program that does more than just *Hello Word!*, the need to use data structures will come up soon. Using structures is even more important if we make one step further and start to talk about Object Oriented Programming.
Structures include arrays, sets and files, but those are not covered in this summary. See https://docwiki.embarcadero.com/RADStudio/Sydney/en/Structured_Types_(Delphi) for details. The focus here is only on records and their various advanced extensions.
There are many different ways to handle these structures in Pascal. There are three main keywords, **record**, **object**, **class** and also other concepts and language elements, like Advanced record, TObject and TClass. Last but not least I add to this list **unit**, as a language element that is rarely seen as a structure to use, but I will show how to use it as such.

In this study I will go through these elements showing what they are and how to use them. As an underlying compiler I refer to FreePascal being the compiler with the broadest hardware and OS base support and being the compiler that can handle most if not all the Pascal dialects available. I will refer to the relevant dialect where applicable.
Most of the development, test I actually did in Lazarus, the most widely used integrated development environment (IDE) for rapid application development (RAD), though for most of the topics covered below the native, text-mode IDE of FreePascal or even a basic text editor and the fpc compiler is good enough.

Learning the details summarized here I relied on available documentation from various releases of Turbo Pascal, Delphi and mainly on the on-line available documentation and wiki of Free Pascal and Lazarus (https://wiki.freepascal.org/) and on the wisdom of the developers and users available in many third party forums, like Stack Overflow (https://stackoverflow.com scattered around many tags, like *freepascal*, *turbo-pascal*, *delphi*, *lazarus*, etc.) but mainly on the dedicated forum of Lazarus at https://forum.lazarus.freepascal.org/index.php. Where applicable, I try to include as many links as possible, to provide more precise descriptions.

3 Record

The mother of all structures used in OOP in Pascal is **record** (https://wiki.freepascal.org/Record). It exists from the very early days and is still very popular and useful. In this chapter I cover the anatomy of it in great details, not only because **record** is so important, but also because it is a great reference point to show the differences of other structural elements later.

3.1 What are records?

Records, or to be more precise **record** type variables are holders of a collection of other variables. As a first approach, the variables, - called fields, - that are stored in a record do not differ much from other variables. From the simple variable types, like integer, to the most complex user defined variable types, practically anything can be stored in a record instance, including variable length types, like arrays or strings. Fields of a **record** can be all different types, unlike in case of arrays and sets where the individual elements are all of the same type.

3.2 Why use records?

If I want to store my name in a program, I can easily use:

```
program Test;
var
  MyFirstName : string;
  MyFamilyName : string;
begin
MyFirstname := 'Zsolt';
MyFamilyName := 'Szakaly';
Writeln('My name is ' + MyFirstName + ' ' MyFamilyName);
end.
```

or I can declare my variable as a record:

```
var
  Myself : record
             FirstName : string;
             LastName : string;
             end;
begin
MySelf.FirstName := 'Zsolt';
```

so, why would I "bother" with a record, use two variable names (OK, only one variable name and a field name) and an extra dot?

The simple answer is that because the two belong together and we need to show it somehow. Besides this simple dialectical approach, using records has many other benefits:

- The fields of a record move together and therefore we cannot corrupt the data within.
Consider that you store the names of students in a class and try to use unconnected arrays vs. arrays of records:

```
var
  GoodApproach : array of
                   record
                     FirstName : string;
                     LastName : string;
                     end;
  BadFirstName : array of string;
  BadFamilyName : array of string;
```

In the good approach, when we add a person and increase the array to hold the name of the person, we can be sure that GoodApproach[10].FirstName and GoodApproach[10].LastName belong to the same person.
In the bad approach, we can easily make a mistake, when adding someone who has no FirstName (think of Madonna) or who has multiple first names and we store them separately. We might end up with two arrays of different sizes and the same index in the two arrays might belong to different persons.

- Records can be transferred together.
Jumping a bit ahead (see next point), but when a record is defined as a type it can be used to move the whole data set together. Think of it:

```
type
  tPerson : record
```

```
    FirstName : string;
    FamilyName : string;
    end;
function GetBossName : tPerson;
```

can work, but the same cannot be achieved if first and family names are stored in different variables.
Similarly, one can define:

```
var
  NameFile : file of tPerson;
```

and read and write full names in one step (beware of fields of a **record** that are actually pointers – see later).

- No need to use long and complicated names.
Names in a certain scope of the program must be unique. If we have in a school teachers, parents and pupils and we want to treat them different than we would need different names, like TeacherFirstName, PupilFamilyName, etc. It is much easier to define teacher, parent and pupil record types and all can have a FirstName and a FamilyName field.
I do not cover here yet (see two points later), but even better would be to use a new type and use it in all the three record types, like:

```
type
  tPerson = record
    FirstName : string;
    FamilyName : string;
    end;
  tTeacher = record
    Name : tPerson;
    Subject : string;
    end;
  tParent = record
    Name : tPerson;
    Profession : string;
    end;
  tPupil = record
    Name : tPerson;
    ClassName : string;
    end;
```

One small remark: Note, that in case of tPupil I have a field called ClassName. It would be much easier to call it simply Class, but **class** is a reserved word in most of the compiler modes of Free Pascal. In case of {$mode fpc} class can be used as a field name, even if it is shown bold in Lazarus editor. Still, I would not recommend to use it, as later you might want to "upgrade" your source to use more features in a different compiler mode and then you would need to change the name.

3.3 Declare as a Variable or as a Type?

At a first glance the same can be achieved defining the record inside the variable declaration or in two steps, first define a **type** and then declare a variable of that newly created type. At the first glance declaring the variable and the record structure in the same step is even easier, shorter and more visible what the given variable stores:

```
var
  TypedVariable1 : tPerson;
  TypedVariable2 : tPerson;
  DirectVariable1, DirectVariable2 :
    record
      FirstName : string;
      SecondName : string;
      end;
  DirectVariable3:
    record
      FirstName : string;
      SecondName : string;
      end;
begin
TypedVariable1 := TypedVariable2;
DirectVariable1 := DirectVariable2;
DirectVariable1 := TypedVariable1;
DirectVariable1 := DirectVariable3;
end.
```

Nonetheless, as a basic rule of thumb always use the two step approach, first define a **type** and then declare the variable as of that type. Again, there are many reasons why one should follow it, with very few exceptions, maybe for very local, short life records.

- The most important reasons can be seen in the above example with red. Those two lines would not even compile, because a variable declared in one step is never the same type as another variable declared literally the same, but in another step or declared in two steps. The only time when two variables declared this way are the same, if they are listed in the same declaration as DirectVariable1 and DirectVariable2 in the above example.

- In the previous point I used a function returning a tPerson type. Functions can return user defined types, but cannot be declared with a return value with a record type defined therein. So, the following would not compile and even if it would, it could not be assigned to any other variable because of the previous point:

```
function FailToCompile : record C : char; end;
```

3.4 How to design the records smart?

Designing a **record** is not an easy task. There is a simple rules of thumb to follow; put together different variables in one record only if they logically belong together, like values describing the properties of an object (properties and objects are used as words of their original meaning, not as Pascal keywords! Pascal keywords are with **bold**). Having said that, there are so many other things to consider, that fully covering it is beyond the scope of this study. Here, I only mention few important aspects, using the example of tTeacher above. We can at least use three different approaches with very similar outcome:

```
type
  tPerson = record
    FirstName : string;
    FamilyName : string;
    end;
  tTeacher1 = record
    Name : tName;
    Subject : string;
    end;
  tTeacher2 = record
    FirstName : string;
    FamilyName : string;
    Subject : string;
    end;
  tTeacher3 = record
    Name : string;
    Subject : string;
    end;
```

- Granularity

The second and the third one only differ in how granular we split the name. Does it make sense to split a name into two? Well, as a first thought, it is good to have as precise data stored as we can. If later we want to sort the names by family name, it is handy to have a separate field for that. On the other hand it limits our capabilities, if we want to include a name with prefixes, like Dr. or multiple first names, etc. In a free format string variable of Name we can put any name we want.
I call this problem the billion dollar problem, as large database migrations typically have this problem. When a company upgrades its billing software or when companies merge and they want to combine their CRMs, it can be a nightmare to split an address written in one line into a record of many fields, like Apartment number, Floor, Building, House number, Street name, etc. Typically the other direction is easier; if we have all these details, we can combine them into a one line Address field. The problem is that when we write our software we cannot know, whether the company using it will acquire other companies or will be acquired. At the end of the day, database migration is a nightmare and companies spend millions to make correct migration tools, or loose even more when they fail to do it right. On a worldwide scale I would not be surprise a billion dollar figure a year lost (well, earned by IT people) on granularity problems.

- Hierarchy

The first and second have the same data granularity, but one has a sub-record with the name details while the other is a flat record. Just like for granularity, there is no easy rule for record hierarchy. Using the flat model is definitely easier to use while the program is small and development time is an issue. Very nicely built, but therefore large and complex hierarchies can be hard to use. Imagine when the record is really complex and there are many levels of data.

```
Presidents[41].PersonalDetails.Name.FirstName[0].OfficialOrNickName := ntNick;
Presidents[41].PersonalDetails.Name.FirstName[0].Name := 'Bill';
Presidents[41].PersonalDetails.Name.FirstName[1].OfficialOrNickName :=
ntOfficial;
Presidents[41].PersonalDetails.Name.FirstName[1].Name := 'William';
Presidents[41].PersonalDetails.Name.FirstName[2].OfficialOrNickName :=
ntOfficial;
Presidents[41].PersonalDetails.Name.FirstName[3].Name := 'Jefferson';
Presidents[41].PersonalDetails.Name.LastName := 'Clinton';
```

(just for the record, Bill Clinton was the 42nd president, but the array is zero based.)

On the other hand, overly simplified flat models can be hard to maintain and use later. Imagine the situation when a company visitor has a record in the database with many fields i.e. a flat record, like first name, last name, who invited, how important VIP is (s)he, etc., and we want to send a request to the card printing vendor to make a visitor card. We cannot send the full record with lots of confidential data, we only need to send the name and the areas to grant access to. If the record is flat, we need to cherry pick the right fields, what might be complicated after a while. In a nice hierarchical structure, we can just send a sub-record what contains exactly what is needed for the card to get printed.

- Naming

Intentionally, I used a misleading type name above. In its current form tPerson only includes name details. A person has many more properties we should use. Either we increase the tPerson with more values in a flat fashion or we use a hierarchical model. Imagine, we want to include the person's age (again, you might get used to the fact that I intentionally use silly examples, so I can correct them later…). If we use the flat model, tPerson is a good name with all the personal details. But if we want to use the hierarchical model, we need to give meaningful names:

```
type
  tName = record
    FirstName : string;
    LastName : string;
    end;
  tPerson = record
    Name : tName;
    Age : integer;
    end;
  tTeacher = record
    PersonalDetails : tPerson;
    Subject : string;
    end;
```

- Database matching
At the end of the day, all serious software has to store the data somewhere, typically in a database. When a physical database is designed, speed and memory usage has to be optimized at the same time. This requires to normalize the database records, without "over-normalizing" it.
To make read and write operation easy, it is often beneficial to use the same structure in memory as the structure in the database, although it is not a must. If there are good reasons, the two can differ.

- Store unchanging values
In the above example I used an Age field for a tPerson. The problem is that the Age of the person is changing every year. You might say, that even the name might change, so what is the problem? Actually the name never changes on its own; someone has to change it. It is an "action" in the software and so the name filed(s) can be updates. The Age on the other hand changes even without anyone asking it.
So, as a good design practice, never store any value that can change without a user interaction. Instead of Age, use Date of birth and if Age is needed, make a **function** (later I also cover **property**) to calculate the Age from DoB and the actual date when the Age is needed.

3.5 Where are the records stored?

As a beginner answer, it should be enough to say that somewhere in the memory. What is more important to know, is that a record instance is immediately created when a variable of that type is declared. Creating the record sounds fancy, but actually in case of **record**s it only means that the necessary memory is reserved for the variable. Important to know, that the size of the variable is always fixed. Actually, you can inquire it in two different ways, through the type and through the variable.

```
type
  tRecord = record
    A : integer;
    B : char;
    end;
var
  Record : tRecord;
begin
writeln('Size of the type ',sizeof(tRecord));
writeln('Size of the variable ',sizeof(Record));
end.
```

will print twice the same number. What this number is, will be covered in the next point.

On a slightly more advanced level, it is known that **record**s are created in the stack, while for example, later on, we will see that **class**es are created in the heap. Nonetheless, we have to be careful with this statement.
First, it assumes that words stack and heap have a meaning in our operating system, while some small special devices (and Pascal can be used to program them) have special memory management systems, so it is only true in the "classic" world of Windows, Linux, Mac, etc.
Secondly, even in the simplest case it is not true, when we declare global variables, those are stored in the heap.
Thirdly, I think officially, unless I make a mistake, there is no rule how a Pascal compiler should store variables. Any new release, or version for a special hardware can do it differently. It is normally not for the programmer to know.

In an even more advance level, one should do some more research and learn more tricks. Even if we think, generally correctly, that local variables including record type variables are stored in the stack, there are cases when it is not so. Not only the compiler might do it like that, but we can also make tricks.

```
var
  Allocation : packed array [1..sizeof(tRecord)] of byte;
procedure LocalInHeap;
  var
    Record : tRecord absolute Allocation;
  begin
  end;
begin
LocalInHeap;
end.
```

3.6 How big is my record?

It is hard to say from just looking at the source code. The best is to ask the system, using the sizeof() function.
What we can know for sure, is that the fields of a record are stored in the memory in the same order as they are declared in the record definition, but they are not necessarily follow each other immediately. Depending on the processor (e.g. 32 bit vs. 64 bit) the fields are normally aligned to a multiply of 4 or 8 or even 16 bytes (https://forum.lazarus.freepascal.org/index.php/topic,56463). This is to make atomic processor operations faster. If it is important not to loose space in between fields, one can use the **packed record** definition, though it can significantly slow down the operation.
The other frequently asked question, what happens with variable length fields? How can the record have a fixed length if its elements are variable length. The "classic" Pascal variable types are always fixed length. The oldest **string**, nowadays called **shortstring** is always 256 bytes long, regardless how many characters are in it (this is why it has a maximum length of 255 characters). All more modern solutions are actually pointers internally and so they occupy the size of a pointer in the record, e.g. 8 bytes in a 64 bit processor. If the variable size field, e.g. **ansistring** is initialized, it only makes the pointer of it to point to the right structure in the heap.

3.7 Record operations

Records are relatively simple creatures and therefore they behave like many other simple variable types. You can use, e.g.

```
var
  A, B : tRecord;
begin
```

```
A :=B;
end.
```

where B is then fully copied to A. This is true for all the fields of record B as it is. In the previous chapter I described that some seemingly simple references are stored as pointers. In such case normally only the pointer is copied, what can cause problems if not handled properly.
Luckily enough some of the most frequently used complex creatures, like **ansistring**, have a copy operator, so although **ansistring** under the hood is a pointer, when used as a field of a **record**, it behaves the same way as a **shortstring**, i.e. seemingly fully copied. Still, we do not need to worry that it copies large strings unnecessarily, as the special reference count mechanism can ensure that it is only copied, if one of the users tries to change it.

This copy mechanism is also one of the reasons why people do not recommend to use it for more complex structures. Every time a **record** is used as a method's parameter not with the **const** or **var** keyword, the call creates a full copy. Copying large records frequently can slow down the software.

3.8 Record as a pointer

There is nothing to stop us using pointers to point to record structures.

```
type
  tRecord = record
    A : integer;
    end;
  pRecord = ^tRecord;
var
  RecordPointer : pRecord;
begin
RecordPointer := new(pRecord);
RecordPointer^.A := 123;
end.
```

This is also a time when our record ends up in the heap, although it is a bit unfair to say so, since we did not declare a variable with tRecord type.

One thing that make people do not like pointers used for record definition is the use of the ^ after the end of the name of the pointer variable. I think that this is the least of the problems with it. It is much more serious that allocating heap objects can easily lead to memory leak if not programmed correctly. This is why I would not recommend to use pointers when we can avoid them. But if we use them, ^ is actually a good reminder to pay attention to memory leak.
For those who want to use pointers for this purpose and still do not like ^, there is good news. In fpc there is a mode switch, you can turn on to automatically de-reference a pointer. With {$ModeSwitch autoderef} the last line of the above example can easily be replaced with

```
RecordPointer.A := 123;
```

but then again, I would not recommend it, if you are afraid of memory leak.

3.9 Variable records

It often happens that for similar objects we want to define one **record** structure, but not all the values we want to store, are relevant to all the variants of the record. A simple approach would be to list all possible variables as fields in the declaration and always use only those that are relevant for the given sub-type. Actually this is possible, but has some drawbacks. First, it wastes memory what was a problem in the past when memory was always of short supply, and secondly, one can add

both values when only one would be allowed creating confusions later. To overcome this problem variable records are allowed to be defined.
In the example above we defined three different **record** types for three different type of people seen in a school. If somewhere else we want to do the same thing with their data regardless of their type, it is better to use a variable **record** to describe them:

```
type
  tSchoolPerson = record
    PersonalDetails : tPerson;
    case integer of
      1 : (Subject:shortstring); // the teacher
      2 : (Profession:shortstring); //the parent
      3 : (ClassName:shortstring); //the pupil
    end;
```

There are a couple of things to be noted:

-The variable part uses a **case** structure with no variable inside the **case** instruction. That is not allowed in a normal **case** statement, but here it is OK. On the other hand if it adds value, there is nothing against using a field identifier also in the **case** statement. In that case the field name given there, is simply an extra field in the fixed part of the structure. In the example shown in the Free Pascal wiki for Records, the following two versions both create the same structure:

```
type
  tMaritalState = (unmarried,married,widowed,divorced);
  tPerson1 = record
    //CONSTANT PART
    //of course records may be nested
    name : record
      first,middle,last : string;
      end;
    sex : (male,female);
    //date of  birth
    dob : TDateTime;
    //VARIABLE PART
    case maritalState : TmaritalState of
      unmarried:();
      married,widowed:(marriageDate:TDateTime);
      divorced:(marriageDateDivorced,divorceDate:TdateTime;
isFirstDivorce:boolean)
    end;
  tPerson2 = record
    //CONSTANT PART
    //of course records may be nested
    name : record
      first,middle,last : string;
      end;
    sex : (male,female);
    //date of birth
    dob : TdateTime;
    //marital status
    maritalState : TmaritalState;
    //VARIABLE PART
    case integer of
      1:();
      2:(marriageDate:TDateTime);
      3:(marriageDateDivorced,divorceDate:TDateTime;isFirstDivorce:boolean)
    end;
```

- The **case** statement, i.e. the variable part, is after all the fixed fields and it has no closing **end;** it is not needed, the closing **end;** of the record definition also closes the **case** structure.

- The variable fields are in a pair of brackets, and not in between **begin-end**, like a **case** statement used in the code.

- In the second version, for married and widowed I did not use two different values, as it does not carry any information. Basically the listing of values in the **case** statement are only to split the different versions of the variable part. It is the case even when the **case** has a variable included. Even in the tPerson1 version, it is not necessary to list both married and widowed in the **case** statement.

- As a consequence or maybe the reason of the above, it is also true that the field in the **case** statement (if any) is not checked at all when the variable part is used. So, even in tPerson1 we can set maritalStatus to unmarried and still set the value of marriageDate.

- The fields of the variable parts occupy the same memory area. After the fixed part the allocation of memory is continued for the first **case** branch. The fields in the second **case** branch also start right after the fixed part. Since different versions of the variable parts can have different number and size fields, the different variable branches can have different overall length. The size of the **record** is as large as needed to store even the largest version.

- Changing a variable part field changes the relevant memory area. In a second step we can read a field from another variable branch and it reads the memory according to the mapping of that definition. Sometimes it can result in totally unpredictable results (imagine a pointer and a double at the same memory address), but sometimes it can also be useful, like when we split a 32 bit cardinal into two 16 bit words.

- In my own example I changed the fields in the variable part from **string** to **shortstring**. The reason for that is that in most of the compiler modes **string** refers to **ansistring**, a special solution as mentioned above. **ansistring** requires initialization that is normally done automatically when created. However in a variable structure the compiler cannot know whether we want to use that variable part or another. Therefore, if we need to use variable types that require initialization, we need to place them outside the variable part. Since, these types are internally pointers, it is not much of a waste to define all versions next to each other.

3.10 Cross referencing

In real life we often need to link instances of different type of records together. Imagine a tTeacher has a link to the best tPupil in their classes and for tPupils they have a class master to link.
So we would want to do something like:

```
type
  tPupil = record; forward;
  tTeacher = record
    TopPupil : tPupil;
    end;
  tPupil = record
    ClassMaster : tTeacher;
    end;
```

Unfortunately, the red line will not work. There is no such thing as a **forward** declaration of a **record**. It is logic, if we think about it. What would the above code mean if we declare a Teacher as a tTeacher, it would allocate the memory of a tTeacher, but that includes a tPupil, so the size of the tTeacher include the size of a tPupil. But the tPupil includes the size of a tTeacher and it never ends.

If you want to use a cross reference, it has to be based on usage of pointers.

4 Advanced record

Advanced records, or also called extended records are a rarely used feature of Pascal. It was added by Embarcadero for Delphi and Free Pascal includes them since version 2.6. See (https://wiki.lazarus.freepascal.org/FPC_New_Features_2.6.0#Advanced_record_syntax, https://www.freepascal.org/docs-html/ref/refch9.html#refse61.html, https://docwiki.embarcadero.com/RADStudio/Alexandria/en/Structured_Types_(Delphi)#Records_.28advanced.29) for official specifications. Experts say, that historically Embarcadero wanted to remove **object** in favor of **class**, but when they realized that sometimes stack structures have a benefit over heap based **class**, instead of revive **object**, they developed **record**. In FreePascal both advanced **record**s and **object**s work.

As advanced records are added late, if I followed a chronological order, they would probably end up somewhere around the end of the list. Nonetheless, I think feature-wise they fit perfectly in between the "classic" **record**s and **object**s.

Advanced records are special in many ways. They do not have their own keyword, the same **record** keyword can be used as above. Why are they called then separate, why not just say that the "classic" **record** has some new features? The reason is not very convincing, but advanced records are not available in the normal compiler modes unless another mode switch is added {$modeswitch ADVANCEDRECORDS}.

Many people consider advanced records unnecessary to use. They say, and they have a point, that for simple reasons one should use **record**, while for more feature rich structural elements one can always use **object** or even more recommended, **class**. Still, there are cases when advanced records can have a place in the programmers' tool set. It does not need to be explained why advanced records are better than "classic" records, but it is beneficial to understand why they are sometimes better than objects or classes. This is also covered in the chapter, but first we check what are the advanced features added to the features of the "classic" brother.

4.1 Methods

When we want to do something with a variable, including **record**s, we can call a **procedure** or a **function**, with the variable as a parameter.

```
writeln('My integer and string are ',i,' and ',s);
```

writeln in this set-up is a general method that can be called with many different types of parameters. On the other hand, if we want to print the personal details of a person, we use a special method, designed to work only with that type of data, including probably a photo, pre-set sections, etc. That method cannot be re-used to print the details of a car. So, what we can do, we can have two different procedures to print them:

```
procedure PrintAPerson(aPerson : tPerson);
procedure PrintACar(aCar : tCar);
```

In a large program this makes a lot of procedures hanging around with specific names, that can only work with a certain type parameter. Things can be made a bit easier, if the same name is used for the procedures (polymorphism), but OOP came up with a better approach, allowing methods to be attached to structures and so we can use:

```
MyCar.Print;
Myself.Print;
```

Besides the obvious benefits of no need to call the method with a parameter there are other benefits.

We do not need to use the variable name inside a method to access it. You should imagine that a method of a **record** has one extra parameter, basically the variable it is called with, and the whole body of the method is included in a **with** statement. Something like as below:

```
type
  tPerson = record
    Name : string;
    procedure Print;
    end;
var
  Person : tPerson;
procedure tPerson.Print;
  begin
  writeln(Name);
  end;
procedure EquivalentTPersonPrint(aPerson : tPerson);
  begin
  with aPerson do
    begin
    writeln(Name);
    end;
  end;
```

Actually something very similar happens under the hood.

Advanced records support normal methods and special methods for construction, called **constructor**s. Unlike the more advanced **object**s and **class**es, **record**s do not have **destructor**s (see there).

Methods of records can have the same name and different parameter lists (polymorphism).

If you are interested in more details, there is a lengthy discussion in this topic at: http://www.delphigroups.info/2/cf/471305.html.

4.2 Visibility

The addition of methods to a **record** raises another issue. We might want to use some internal variables, fields in our **record**s that are only used by our methods, but we do not want to give others access to them. This requires a new concept, called visibility. In case of advanced records we can use **private**, **strict private** and **public** visibility modifiers. **public** fields and methods are visible from anywhere where the record is visible, **private** is visible only within the same unit and **strict private** is only visible from within the methods of the record.
The default visibility is **public**.

4.3 Properties

Visibility is also nice in a case we saw above, when in the data structure we store the date of birth, but we want the user to give or retrieve the age. We can do it with the following structure:

```
type
  tPerson = record
    private
      DoB : tDateTime;
    public
      function Age : integer;
      procedure Age(a : integer);
    end;
```

We can now do things like:

```
Myself.Age(99);
writeln(Myself.Age);
```

Due to the nice behavior of Pascal, in the second line we cannot differentiate if Age is a field (variable) within the record or a function, unlike in C++, where we would see

```
std::cout << Myself.Age() << " " << Myself.DoB << "\n";
```

With the setting side though we still see that Age is not a field, but it is a procedure. For this problem there is a new solution, called properties. Adding a property for the Age, we can now write:

```
  tPerson = record
    private
      DoB : tDateTime;
      function GetAge : integer;
      procedure SetAge(newage : integer),
    public
      property Age : integer read GetAge write SetAge;
    end;
```

Now we can access this new property fully transparently, as if it were a field:

```
Myself.Age := 99;
writeln(Myself.Age);
```

If a property has only a read mechanism, it becomes read only and the same for write only.

4.4 Class elements

Normally, if we have multiple instances from a record type, we expect that all record instances hold the data of one object, and this is how a normal record works. Nevertheless, sometimes we might want to have a field that is stored as a "global like variable", but still linked to this particular type of record. We might want to store the oldest person's age when we have many records. For this we can declare a so-called class field what is shared across all the tPerson type records.

Few remarks:
- this **class** keyword is not the **class** keyword used later as a more advanced version of a **record**.
- not only fields, but also properties and methods can be **class** level.
- class level fields are not only shared among the instances of tPerson, but already created when the type is defined, and hence can also be used even when no instance is declared, using like tPerson.MaxAge
- class variables can only be in the fixed part, not in the variable part.
- class method declarations are always followed by the **static** keyword. It is not very clear why it is required here, but it is a must. The use of **static** will be discussed in the next chapter for **object**s.
- in order to use **class** in advanced records the mode switch *advancedrecords* is not sufficient, also the compiler mode has to be *delphi* or *objfpc*.

And now an example:

```
type
 tPerson = record
   private
     class var oldestDoB : tDateTime;
     class function GetMaxAge : double; static;
   public
     class property MaxAge : double read GetMaxAge;
   private
     DoB : tDateTime;
     procedure SetAge(aAge : double);
   public
     property Age : double write SetAge;
   end;
class function tPerson.GetMaxAge : double;
 begin
 GetMaxAge := (now - oldestDoB) / 365;
 end;
procedure tPerson.SetAge (aAge : double);
 begin
 DoB := now - aAge;
 if (oldestDoB = 0) or (DoB < oldestDoB) then
   oldestDoB := DoB;
 end;

var
  Person1, Person2 : tPerson;
begin
writeln(Person1.MaxAge);
Person1.Age := 34*365;
writeln(Person2.MaxAge);
Person2.Age:= 54*365;
writeln(tPerson.MaxAge);
end.
```

(Output is not formatted and age is calculated with 365 day years, but I think the use of class elements can be seen.)

As a summary:

- Class level fields are defined by using **class var** in front of the name of the field and no **static** after.
- Class level methods need both **class** in front of the **procedure** or **function** keyword and **static** after in the definition. In the implementation of the method the **class** keyword is also necessary. Class level methods can only access class level fields;
- Class level properties are defined with the **class** keyword in front of their name. Class level properties can only access class level methods and fields in the **read** and **write** section.

4.5 Nested types and consts

Normally constants and new, user defined types can be placed at the beginning of the program or unit as they do not change anyway, do not use extra memory regardless where they are defined. Nonetheless, for visibility (i.e. to prevent that noone else who is not supposed to use it, should not use it) and for avoiding name conflicts (**const** MaxRecords = n) can be used for various types of records with different n numbers) we often put **type** and **const** inside local **procedure**s and **function**s.

The same concept can be applied for advanced records, i.e. inside the record definition **type** and **const** keywords are allowed.

4.6 Benefits compared to object

In the next two chapters I will show **object** and **class** as more advanced structures to use for OOP. As I did in this chapter, I will show there also, what are the new features and improvements compared to the previous chapter, i.e. in case of **object** compared to advanced **record**s.
However life is not always going upwards. There are some aspects, where advanced **record**s can do a bit more than **object**s and **class**es, and so give a reason for existence (other than full Delphi compatibility of FreePascal). So, the following points are worth considering when you have to choose between an advanced **record** and an **object**:

- Advanced records are still records, nice and small, easy to understand.
- Records can have variable parts. It is a feature that disappears in the next steps.
- Advanced records still behave like any other value types, so the copy operator copies the full content as described above.
- On Win32 platform there are some more differences, but for that, please refer to
https://docwiki.embarcadero.com/RADStudio/Alexandria/en/Structured_Types_(Delphi)#Records_.28advanced.29

5 Object

In our logical sequence the next kind of structure that can be used for OOP is **object**. https://www.freepascal.org/docs-html/ref/refch5.html#refse29.html By today, even **object** is considered a bit out of date, although still fully supported and in my view, still very useful.
I think **object** was the original keyword to support OOP in Pascal. If you read the documentation, it mentions that visibility was first introduced in **object**s. Using advanced records came later with many of the **object** functionality introduced for **record**s. As a result **object**s are very similar to advanced **record**s. In this chapter I try to collect what more **object**s can do than **record**. (The other direction, i.e. what **record** can do, but **object** cannot, can be found in the previous chapter).

To **object** keyword is supported even in the basic *fpc* compiler mode, however some advanced functionalities are limited.

5.1 Class variables

In the same way as we saw in case of advanced records, there can be fields that are not stored on the instance level, but on the type level. There we had one way to declare a variable

```
type
  tRecord = record
    class var I : integer;
    end;
```

In case of **object** we have two different methods, with exactly the same result:

```
type
  tRecord = record
    I : integer; static;
    class var J : integer;
    end;
```

The only difference that **static** works in all cases, **class var** requires either *delphi* or *objfpc* compiler mode.

It must be mentioned that the two versions cannot be combined, i.e. we cannot use both **class var** and **static** in the same declaration.

5.2 Class properties

Class properties work the same way as in case of advanced records.

5.3 Class methods

When discussing methods in case of **object**s, the story gets a bit more complicated. There are basically two dimensions to think of, when designing **object** methods. Although in case of advanced **record**s, we did not have that, still used two special structural elements:

```
type
  tRecord = reocrd
    class procedure Test; static;
    end;
```

In case of **object** the two structural elements, **class** in front of **procedure** or **function** and an extra keyword like **static** or **virtual** after the definition might serve two dimensions.
One dimension is, whether the method always works on a given instance or whether it can work independently of an instance, i.e. on class level.
The other dimension is, how the method works when called.

The keyword to make a method a class method, is similar to class level **object** fields, i.e. either **class** in front of **function** or **procedure** or **static** after or in case of methods both of them is also allowed. All the three approach has a very similar result, the method becomes class level and thus cannot access instance fields.

Still there are differences. I would expect based on the documentation of **class** (https://www.freepascal.org/docs-html/ref/refsu30.html) that **static** methods have no **Self** parameter, however for **object** I did not find such documentation. As the below example shows, for **object** it is not even true. Only when both **class** and **static** keywords are used then the method has no **Self** parameter.
The other difference is that when the method is only **class** defined, but not **static**, it cannot be called from the type, only as an instance.
Unless I made some mistake, this is not totally coherent in my view.

A long example to show, what is possible and what is not:

```
type
  tMyObject = object
    // Instance variable
    IInstance : integer;
    // Class variable, two types
    IStatic : integer; static;
    class var IClass : integer;
    // Cannot use both
//  class var IClassStatic : integer; static;

    // Can define both a class and an instance constructor
    class constructor CClass;
    constructor CInstance;

    // All the four declarations are possible
    procedure PInstance;
    procedure PStatic; static;
    class procedure PClass;
    class procedure PClassStatic; static;

    end;

class constructor tMyObject.CClass;
  begin
  writeln('tMyObject.CClass called');
  // Class constructor can only see the class variables, regardless how they
were defined
//IInstance        := 111;
  IStatic          := 112;
  IClass           := 113;
  end;

constructor tMyObject.CInstance;
  begin
  writeln('tMyObject.CInstance called');
  // The instance constructor can see both class and instance variables
  IInstance        := 121;
  IStatic          := 122;
  IClass           := 123;
  end;

procedure tMyObject.PInstance;
  begin
  // Instant procedure has a Self variable and can see all variables
  writeln('tMyObject.PInstance - ',IInstance, ' ' , IStatic, ' ' , IClass);
  writeln('tMyObject.PInstance @ ',Self.IInstance, ' ' , Self.IStatic, ' ' ,
Self.IClass);
  end;

procedure tMyObject.PStatic;
  begin
  // Static procedure has a Self variable, but can only see the class variables
  writeln('tMyObject.PStatic - '(* , IInstance, ' ' *), IStatic, ' ' , IClass);
  writeln('tMyObject.PStatic @ '(* , Self.IInstance, ' ' *), Self.IStatic, '
' , Self.IClass);
  end;

class procedure tMyObject.PClass;
  begin
  // Class procedure has the same access
  writeln('tMyObject.PClass - '(*, IInstance, ' ' *), IStatic, ' ' , IClass);
  writeln('tMyObject.PClass @ '(* , Self.IInstance, ' ' *), Self.IStatic, ' ' ,
Self.IClass);
  end;

class procedure tMyObject.PClassStatic;
  begin
  // Class and static procedure can only see the class variables and has no
Self variable
  writeln('tMyObject.PClassStatic - '(* , IInstance, ' ' *), IStatic, ' ' ,
IClass);
//writeln('tMyObject.PClassStatic @ ', Self.IInstance, ' ', Self.IStatic, ' ' ,
Self.IClass);
  end;

var
  MyObject  : tMyObject;

begin
  writeln('tMyObject');
  // Class constructor can never be called, called automatically
```

```
//tMyObject.CClass;
  // Instance constructor cannot be called from type
//tMyObject.CInstance;
  // Instance procedure cannot be called from type
//tMyObject.PInstance;
  // Static procedure is a class procedure, can be called and gets a type level
Self
  tMyObject.PStatic;
  // Class procedure cannot be called from a type
//tMyObject.PClass;
  // Class and static procedure can be called with no Self
  tMyObject.PClassStatic;
  writeln;
  writeln('MyObject');
  // For instances we can simplify with "with"
  with MyObject do
    begin
    // Class constructor cannot be called, but everything else yes
//  CClass;
    CInstance;
    PInstance;
    PStatic;
    PClass;
    PClassStatic;
    end;
end.
```

5.4 Inheritance

Some of the problems mentioned when we discussed above the dilemma, whether FirstName and FamilyName should be in a separate tName record type and the tPerson record should include tName or FirstName and FamilyName should be direct fields of tPerson can be overcome with the concept of inheritance. A tTeacher is actually a tPerson with some extra fields and methods, tPerson is actually a tName with some extra fields and methods, etc. The new keyword **object** can handle inheritance.
We can now define:

```
type
  tName = object
    FirstName : string;
    Familyname : string;
    end;
  tPerson = object( tName)
    DoB : tDateTime;
    end;
  tTeacher = object(tPerson)
    Subject : string;
    end;
```

Now, if we want to access the first name of a teacher, we do not need to go through a chain of fields, like Teacher.Personality.Name.FirstName, but we can do easily

```
writeln(Teacher.FirstName,' ',Teacher.LastName);
```

as if FirstName was declared directly in tTeacher.

As a first approach we typically want our object types be usable and want them to also serve as a parent type for later types. Nonetheless, in Pascal we can block both of them, but never at the same time.

Using the **abstract** keyword we indicate that the type should never be used, only its descendants. Imagine a tShape object including a method to calculate the area. While we do not specify the shape and coordinates of the shape, we cannot use it. So we can declare tShape as abstract, and the descendant tSquare and tCircle objects can be used only.
Defining an object with the **sealed** keyword we make sure that no descendant type is defined based on this.

5.5 Inherited methods

Besides inheriting variable fields, object hierarchy has another very nice feature, method inheritance. If we define two object types tTest1 and tTest2 and tTest2 is a descendant of tTest1 then any tTest2 instance can safely use any method defined in tTest1. There cannot be any problem, since a method defined in tTest1 can only use fields that are already available in tTest1, any additional fields of tTest2 are simply not seen. Need to remark that this is also the reason, why a variable field cannot be re-declared in tTest2.
There is no restriction though how a method can be redefined. Basically any (non-virtual – see later) method redefined behaves as a totally new method. tTest2.SameMethod can access tTest1 and tTest2 variables and can do anything. There is no restriction regarding the parameter list of the method either, even that can be different in the redefined version.
For a tTest2 instance it is still possible to call tTest1.SameMethod, even if it is redefined as tTest2.SameMethod by simply calling it by putting tTest1 in front of the method name, or from within tTest2.SameMethod by calling the inherited SameMethod by keyword **inherited**.
If tTest1 has two methods Method1 and Method2 and both are redefined in tTest2, it is still an easy case. If a tTest1 instance calls Method1 from Method2, it actually calls tTest1.Method1, but if the same is done from a tTest2 instance then tTest2.Method1 is called.
If from the two methods Method1 is not redefined in tTest2, that is still not a problem. When a tTest2 instance calls from a redefined Method2 a not redefined Method1, there is no other choice just to call tTest1.Method1.
The problem starts if Method1 is redefined, but Method2 is not. We call from a tTest2 instance Method2. As it is not redefined it must call tTest1.Method2, but tTest1.Method2 wants to call a Method1. The difficult question, whether it should call the original tTest1.Method1 or the redefined tTest2.Method1. This I cover in the next point.

5.6 Virtual methods

In the previous point the problem arose which Method1 to call, if we have a call from a descendant tTest2 type instance calling a method called Method2, what is only defined in the parent tTest1 **object** definition, but not redefined in the descendant tTest2, and subsequently it calls a Method1, that is originally defined in tTest1 and then redefined in tTest2.
Logically a tTest2.Method2 can only turn to its own declaration when starts looking for Method1. Unless we do some special efforts, it is logical that tTest1.Method2 will call tTest1.Method1, even if originally we called it from Test2. In this case we simply loose the access to tTest2.Method1 from tTest2.Method2, since the latter does not exist and the ancestor's tTest1.Method2 is used. This is the default way.
In most cases however we want to use the "closest" version of every method. Imagine a hierarchy of tTest1, tTest2, tTest3. When we do something on an object instance of tTest3, we want to use the tTest3 methods if available, or the tTest2 if only that is available and if none of them is there, then we shall use tTest1. This is true even if some in-between methods are not available.
The solution is to make a method **virtual**. In the above example tTest1.Method2 looks for a Method1 in its own declaration. If **virtual** is not found, it uses that, but if it finds **virtual** then it knows to use the one that is "closest" to the actual object instance.
How can it find that? It is rather simple, the object passed to Method1 has a hidden table showing the entry points of all the virtual methods it is aware of. If tTest3 redefined a method then the table

points to that method, if it was only redefined in tTest2, but not in tTest3 then the table points to the method defined in tTest2 and if none of them redefined the original tTest1 method then it points to that.

Few words to pay attention to:

- Since tTest1.Method2 does not know at the time of compile whether its Method1 will be overwritten later or not, it can only call it with the know parameter list of tTest1.Method1. As a consequence it is not allowed to redefine a virtual method with a different parameter list than the one in the ancestor.

- If the **virtual** keyword is missing in tTest1.Method1 then it will never look for a virtual table in the object instance, hence still the parent's method will be called.

- If Method1 is redefined in tTest2 and tTest3, but we forget **virtual** from the third declaration, it has a strange effect. When tTest3 is compiled, it creates a new static Method1 visible from tTest3 only, but it still maintains a virtual table inherited from tTest2 and in it there is a pointer to a Method1, to actually the one defined in tTest2. So the unwanted effect that Test3.Method2 will call tTest1.Method2 and that will call tTest2.Method1.

- At one point in history, Delphi developers were concerned that in case of complex object hierarchies the virtual tables can grow large and it can have a memory shortage consequence. Therefore another keyword was introduced, called **dynamic**. Methods declared **dynamic** behaved the same way as **virtual**, only the virtual table was not stored, but created on the fly. That resulted less memory usage, but slower execution. Free Pascal still supports the **dynamic** keyword, but it is equivalent to **virtual**.

- Although an object is immediately created when it is declared, just like records earlier, the virtual tables make it a bit more complicated. Virtual tables need to be initialized, if used. The good news is that the system automatically takes care of it, if the object or one of its ancestors has a **constructor** (see next point) and it is called explicitely.

- Class methods can also be virtual. In that case to make it class level, only the **class** keyword can be used before **procedure** or **function**, not static after. This is simply, because **static** and **virtual** cannot appear in the same definition.
My private thought is that it would have been probably better, to clearly separate the two dimensions mentioned above, and always use only **class** for making a field or method a class level one and use the keyword after the definition to specify how it is inherited, like **virtual**, **dynamic**. In this context **static** would not be necessary but if used, could mean the default inheritance behavior. But again, it is not the case, as **static** has also a different method calling signature without Self.

5.7 Constructor and Destructor

In case of advanced records, I already mentioned briefly the keyword **constructor**. A method defined as **constructor** has the same format as a **procedure**, the only difference is that the keyword **procedure** is replaced by **constructor**. The difference is that when a **constructor** is called, it sets up the virtual table without any extra call to be made.
Constructors therefore can be totally empty, but still has to be called if the object uses **virtual** methods.
In real life, most of the time constructors are also used to initialize variables, etc. and therefore constructors often have parameters as well. One object definition can include multiple constructors with the same or different names. Obviously, if they have the same name (Init is typical), they have to have different parameter lists, no two identical name and parameter list method can exist.

The opposite of **constructor**s are **destructor**s. When we finished with an object, we should call its **destructor** to free up memory reserved for the **object**. Unlike **constructor**, one object should only have one **destructor**, otherwise a compiler warning is issued. Most of the time **destructor**s are virtual, so always the most relevant is called.

Constructors and **destructor**s can also be made class level, i.e. belonging to the object type rather than to an **object** instance. There can only be one **class constructor** and **class destructor** what are called automatically. The constructor is called before the **initialization** section of the **unit** where it is defined and the destructor is called after the execution of the **finalization** section.

5.8 Memory usage

Unlike in case of records it is not officially guaranteed that fields of an object are stored in the same sequence as they are declared. The compiler might use some optimization if prefers.

It is often forgotten however, **object**s still behave in many ways similar to **record** and so, we can use the **packed** keyword, what does exactly the same as in case of records, i.e. fields are stored in the same sequence as declared without any padding, address alignment, or with other words, no holes in between them.

5.9 Heap or stack?

Earlier we already touched this topic. Normally a local variable **object** is stored in the stack, just like any other variable, including **record**s. All the fields get their memory allocation and the object can be used, if no virtual table is needed. Otherwise, it can create a segmentation fault or similar when it is used.
As a general rule, it is recommended therefore to always call a constructor. But if a constructor is called anyway, it is easy to refer the object by reference (basically pointer) instead of a variable. In this case the object gets its memory in the heap, making it even more important to call its **destructor**, once not needed.

The two type of usage is shown below:

```
type
  tMyObject = object
    constructor Init;
    destructor Done; virtual;
    procedure Print;
    end;
 pMyObject = ^ tMyObject;

var
  Variable : tMyObject;
  ReferenceNoInit : pMyObject;
  ReferenceInit : pMyObject;

...

Variable.Print; // works, as no virtual methods used, has compiler warning
Variable.Init; // unnecessary in this case
Variable.Done; // also unnecessary

ReferenceNoInit^.Print; // would fail if Print uses any field of tMyObject as
the object is not yet created
```

```
ReferenceNoInit := new(pMyObject); // OK, but does not call a constructor, 
compiler warning
ReferenceNoInit^.Init; // the constructor can be called later

new(ReferenceNoInit); // the same as above, no constructor call, compiler 
warning

ReferenceInit := new(pMyObject,Init); // The correct way to create a heap 
object
new(ReferenceInit, Init); // Equivalent to the above
```

Please note that I called my object type tMyObject and not TObject, as that is a pre-defined **class** in FreePascal, as shown in a chapter later.

As a reminder I can mention here as well, that there is a mode switch allowing auto dereferencing of the pointers, so the ^ sign is not always needed, albeit I do not recommend it.

5.10 Fail

Inside **constructor**s a rarely used instruction can also be used. That is **fail**: https://www.freepascal.org/docs-html/rtl/system/fail.html. What fail does is similar to **exit** with a significant difference. The execution of the constructor is terminated and the program returns to the caller, but the virtual table is not created.
What happens to the actual object instance, depends on the way we created it and how we want to use it.

If the **object** is a variable type, then all the memory is allocated when the variable is declared, i.e. the object instance is "created". When the **constructor** is called, it runs up to the point of **fail**. Any activity done before **fail** is still there, e.g. if a field is set, it remains like that. The object instance can be used normally as long as no virtual method is involved. If a virtual method is called, the lack of virtual table results in a crash, similarly to the case when the **constructor** is not called at all. This is the difference in comparison with **exit**. If the constructor has an **exit** in it, the virtual table is still created.

If the object instance is created in the heap, we still have two cases.
The bad programming practice as shown above to allocate the memory with a **new** and in a second step call (if not forgotten) the **constructor**. Since at the moment of calling the **constructor**, the memory is already allocated, the pointer is set and **fail** does not reverse these. As a result, the object instance is still more or less working, just like in case of a variable type **object**, except the fact that virtual methods cannot be called.
The correct way to create a heap object though, as also shown above, is to call the **constructor** in the extended syntax of **new**, in the same step. If the **constructor** then fails with **fail**, then no memory allocation happens and the object is simply not there. The pointer variable or reference to the object instance will be **nil**, and the object cannot be used (except class methods that do not use any instance fields).

5.11 Visibility

In case of advanced records we saw that visibility of methods can be set to **strict private**, **private** and **public**. As a reminder: **strict private** is only visible from inside the **record**, **private** is visible inside the **unit** and **public** is visible from anywhere outside of the **unit**.
In case of **object**, things get a bit more complicated. Two new visibility levels are introduced, **protected** and **strict protected**.
The more open **protected** means that fields and methods in that section are available for anyone within the same **unit** (like **private**) and for descendant types anywhere else outside of the **unit**.

The less open **strict protected** means that fields are available for the type and all its descendants anywhere, but not to others even in the same **unit**.

6 Class

Today, the most advanced structure to be used for OOP in FreePascal is **class** https://www.freepascal.org/docs-html/ref/refch6.html#x69-930006.
The history and naming of **class** vs. **object** is of a long one and Delphi played a major role in this development. Most experts say nowadays to use only **class** in Delphi https://stackoverflow.com/questions/6103747/is-there-a-benefit-in-using-old-style-object-instead-of-class-in-delphi and even in the FreePascal community the mood is to use **class** or **record**, but seldom **object**. I still do not fully agree with the latter, as **object**s are nice, small and easy to understand, unlike the more complex **class** types.
In this chapter I collect what the **class** keyword brought to us in Pascal. The good news that **class** works identically in most of the Pascal implementations and FreePascal compiler modes (if defined).

Also there is a very good discussion on the Lazarus forum in this topic, highlighting the evolution of record-object-class-advancedobject in Delphi and Free Pascal.. I especially recommend the linked comment with a comparison of them:
https://forum.lazarus.freepascal.org/index.php/topic,30686.30.html.
Also a similar table is available on Wikipedia:
https://wiki.freepascal.org/Record.

6.1 Class instance vs. object

A small remark on a generally used, but confusing terminology. For records, advanced records and objects I tried to use phrases clearly; when I talked about e.g. an advanced record type, and when about an object instance. I tried to avoid using a word like object without mentioning if I meant the type or the instance. Using the word object to me was either the keyword **object**, or a general word describing something, not in a programming meaning.
Similarly, moving to the new keyword **class**, I will try to continue to use a class type and a class instance. Especially in the Delphi environment, where **object** is not really a supported structure, but also in the FreePascal world, people often call a class instance simply an object. I would recommend not to do so.

6.2 Pointer or not?

In case of **object** I lengthily discussed when an object instance is created as a variable (a.k.a. in the stack) and when as a reference (a.k.a. in the heap). The thing is that **class** is always behaves like the second approach, although it disguises itself as it it were the first.

```
type
  tMyClass = class
   end;
var
  MyClassInstance : tMyClass;
```

looks like that MyClassInstance is a variable, i.e. behaves as Variable behaved in 5.9. The truth is that MyClassInstance is actually a reference (or pointer) and so it is more similar to ReferenceInit from that example. The last line should be understood as MyClassInstance : ^tMyClass but using the **class** keyword the compiler makes this automatically for us.

Similarly when we use a given class instance, being a reference, we should use MyClass^.Print. We already saw earlier that with a mode switch we can make pointer dereferencing automatic. The keyword **class** also makes it automatic for us, so we can use MyClass.Print even when we know by now that MyClass is only a reference.

If you have doubts, you can always print out sizeof(MyClassInstance) and you will get back the size of a pointer depending on your system and not the actual size of all the fields, virtual table, etc. reserved when a class instance is created.

Please note, that just like in case of **object** where I did not use tObject for my definition, here again I did not use tClass. The reason is the same; TClass is also already used by FreePascal for other purposes, covered later.

6.3 Must create

We saw that if we want to create an **object** instance in the heap, we need to reserve the necessary space with **new**, and as a general rule also always initialize it with a **constructor**, preferably in the extended syntax of **new**, or in a separate step right after.
The new structure defined with **class** is very similar to **object** defined in the heap. As a consequence a class instance must always get its place in the heap and because its root ancestor (Tobject, see later) already has some virtual method, it also must be initialized. To do these two steps in one instruction, **class** uses a new syntax:

```
MyClassInstance := tMyClass.Create;
```

where Create is a **constructor** method's name.

As in case of object, **constructors** are rather flexible to define:
- A class type can have multiple constructors.
- Multiple constructors can have different names.
- Multiple constructors can have different parameters, even with the same name.

Constructors can have **fail** in it and it works as described in case of **object**s when used in the extended syntax of **new**, i.e. the necessary memory is not reserved and the pointer is returned as **nil**.

Please note, that while in case of **object** I used constructor Init;, in case of **class** I used constructor Create;. In principle we have absolute freedom what name to use for **constructor**s in both cases, the words Init and Create describe better what they actually do.
In case of **object**, the "normal" usage is to declare the instance as a variable and therefore it is immediately created in the memory. The purpose of the **constructor** is simply to initialize any user defined fields and the virtual table. Hence the name Init.
In case of **class**, the constructor is used both to reserve the memory and to initialize it, hence the different name showing that it is not only initialized, but fully created in this step.

Theoretically, we can have a class type variable that we do not create with a **constructor** and can still use. This is if we use only class methods and fields, however if this is the purpose, it is better not to declare a variable at all, but call the methods with tMyClass.Method logic.

6.4 A class type is never empty

Unlike **record** and **object** keywords that describe some language structures only, **class** is different. There is a top ancestor definition called TObject and all **class** definitions implicitly inherited from it if we use an empty class definition. The following two declarations are the same.

```
type
  tMyClass1 = class
    end;
  tMyClass2 = class(TObject)
    end;
```

Even if we use a parent class type then that has TObject in it, since no first class type other than TObject can exist. It is not allowed to define something as tEmptyClass = class().

The details of TObject I cover in the next chapter, but it is important to mention here, that it is beneficial to use many of the predefined features of TObject and also to "respect" some of the rules that are set-up there, even if not mandatory (see Destroy in the next point).

6.5 Must destroy

A class instance is always in the heap even if the variable holding the pointer (reference) is a local variable stored in the stack. The problem comes when a procedure or function creates a local variable class instance therein and it is not destroyed before the method exits. The pointer is lost, but the memory is still reserved. If it happens many times, we might run out of memory.
Therefore it is very important to destroy the class instance with a **destructor** once not needed.

As for **constructor**s I mentioned the naming convention of Init and Create for **object** and **class** respectively. The similar naming practice is in use for **destructor**s, using Done and Destroy respectively. Furthermore, class types are all descendants of TObject, and there the **destructor** is called Destroy. As shown below, the best way to free an object instance properly requires that the **destructor** is called Destroy with **override** (also discussed later).

The simplest way is or would be to destroy a class instance by calling its **destructor**: MyClassInstance.Destroy;, or whatever name we give to it. However it is not recommended.

As said above, a class type is always a descendant of TObject and TObject has already got a **destructor** called Destroy and declared **virtual**. Calling that or an overridden version of that or any other destructor defined later carries a risk. If the class pointer (i.e. the class variable) does not exist Destroy can crash. Therefore TObject has a method called Free that first checks if the instance exists and only calls Destroy if it does. Therefore the correct way to destroy an instance is to call

```
MyObject.Free;
```

It has to be mentioned that Free does not set the variable to **nil**, i.e. assigned(MyObject) can still return true after Free is called. If it is not the last step before the variable goes out of scope and has the chance that later somebody tries to check or use it, it is better to call a standalone procedure called FreeAndNil(MyObject). That also calls MyObject.Free and sets it to **nil** in any case, even if the Free fails for whatever reason.

This is however a dividing topic: https://stackoverflow.com/questions/3159376/which-is-preferable-free-or-freeandnil among programmers.

In many cases we want to define our own **destructor**, e.g. to call files, release resources, etc. As said above, we can define our **destructor** as

```
destructor tMyObject.KillAndDelete;
```

and can call it

```
MyObject.KillAndDelete;
```

it is strongly not recommended. If we use

```
FreeAndNil(MyObject);
```

or simply

```
MyObject.Free;
```

it will call internally the Destroy method. If we therefore want our **destructor** to be used, it has to be called Destroy and it has to be declared to be virtual. Below I will cover more in details the virtual methods of a **class**, here it should be enough to mention that the keyword to use is not **virtual** as in case of **object**, but **override**.
Since on all ancestor levels there might be things to be destroyed, it is very important to call the **inherited** Destroy every time.

6.6 Really must destroy

Memory leak is a serious problem. Large corporations struggle with memory leak problems in large software or firmware of practically any sort of device. It is so common, that many devices have a regular restart programmed in every day or every week because the developers could not ensure that the program is memory leak free.
Besides the "amateur" mistake of simply forgetting to release a memory blocked or destroying a class instance completely, the most common reason of a memory leak is that in certain cases a method exits elsewhere than at the end and thus the **destructor** of a class instance is not called.

The two frequent cases are when the method has multiple exit points, using the **exit** command or when an **exception** is raised.

The first is a program design question. Some large corporations even ban multiple exit points completely for easier maintenance of their code and to avoid this problem. If it is not forbidden, still a very close attention is needed to release every memory and instance that was blocked or created before the **exit** is called (but do not try to release one that was to be created later in the code).

The other is even more difficult.
If our own code in a method raises an **exception**, the same method should never try to catch it. If we can solve it, it is not an exception! So, if we raise in a method an exception, it shall go uncaught and that is practically equivalent with an immediate exit. If we blocked memory or created an instance before that point, it remains there and if the whole program is not crashed (i.e. the exception is caught somewhere higher in the program hierarchy), it is a memory leak. Since we are in control of raising our own exceptions, obviously we can pay close attention just like in case of **exit** to release the memory and instances before the **exception** is raised.
If the exception is raised by a method or operation called from our code, including a simple division by zero, we either catch it or not. If we catch it in a **try..except..end** structure then we are safe, our program continues after the **try..except..end** and we can still safely release the instance.
The most hidden problem, if we do not think that a method or operation called can raise an exception and it goes unnoticed, raising it even higher. This means however that our method exits just as if it raised the exception, but we do not handle it properly.

For all these cases, except the one when we immediately handle a potential exception raised by a called method or operation, there is a solution. That is the **try..finally..end** structure. Regardless why the try section is finished, let it be an early **exit**, an **exception** raised directly or indirectly therein, or the section just finishes normally, the **finally** section will always run. Therefore it is strongly recommended to include a **try..finally..end** structure after the creation of any object:

```
procedure test;
  var
    MyObject : tMyObject;
  begin
  MyObject := tMyObject.Create;
  try
    MyObject.DoAnything;
    …
  finally
    FreeAndNil(MyObject);
    end;
  end;
```

6.7 Virtual methods

I thoroughly discussed how virtual methods are used in case of an **object**. However some reader might have detected a small and not very important loophole in it. I try to explain it.

Lets assume we have a whole chain of **object** declaration from tTest1 to tTest4 and all of them have PrintToPrinter and PrintToScreen method. The highest level tTest1 has also got a Print method, that asks where to print, to a printer or to the screen and calls the appropriate one from the two. If we have e.g. a Test3 : tTest3 object instance then we want Test3.Print to call tTest3.PrintToPrinter even if Print is only declared for tTest1. Therefore we declare in all the four levels of object types the PrintToPrinter method virtual.

We saw, that if we forget to define tTest3.PrintToPrinter virtual then Test3.Print will call tTest1.Print and that will call the closest virtual PrintToPrinter, i.e. tTest2.PrintToPrinter.

Now, if tTest3 has a new method tTest3.SmartPrint that can also call PrintToScreen, we face another problem. We want a Test4 : tTest4 instance to call Print and SmartPrint, but we want that if it calls PrintToScreen through Print, it should use tTest2.PrintToScreen (end of the chain started in tTest1), but if it calls SmartPrint it should use its own tTest4.PrintToScreen.

I know it is not a typical use case, but I could not come up with a better. The important point is that this cannot be done
If in all the four levels we declare PrintToScreen as **virtual** then the call works with the following logic: Test4.Print calls the only Print available, i.e. tTest1.Print. Then tTest1.Print wants to call a PrintToScreen. Since PrintToScreen is virtual on all the four levels, it will use the closest one, i.e. tTest4.PrintToScreen. This was not our goal., for Print we wanted to use the end of the first chain.
If for tTest3 we do not use the **virtual** keyword for PrintToScreen to stop the chain, then the call works as follows: Test4.SmartPrint calls the only available SmartPrint, i.e. tTest3.SmartPrint. Then tTest3.SmartPrint wants to call a PrintToScreen and it finds tTest3.PrintToScreen and since it is not **virtual**, it stops there and uses that. Again, this was not our goal.

To summarize in one sentence it is not possible to terminate a virtual chain and at the same time start a new one.

This was all for the **object** keyword. For **class** it works differently. The first method in the virtual hierarchy must be declared with the **virtual** keyword. Any subsequent descendants in the same chain must be defined with the **override** keyword. We can still terminate the chain by not using

neither the **virtual** nor the **override** keyword as for an **object**. Here we have a third option, to use again the **virtual** keyword and that terminates the previous chain an immediately starts a new one.
In our previous example we would define tTest1 and tTest3.PrintToScreen **virtual** and tTest2 and tTest4 **override** and would get the result we wanted.
If the virtual chain is interrupted by adding **virtual** again to it as we did in this example or it is simply interrupted by not adding **override** to it, the compiler issues a warning. To indicate that it was indeed our intention we have to add another keyword **reintroduce** to suppress the warning.

This is the reason, why Destroy should always be declared with **override** and not with **virtual**.

6.8 Class and instance elements

The concept behind class level elements is the same as seen with **object**, with few differences.

Class and instance level variables and properties work the same ways as in **object**.

Methods are slightly different.
- The instance **constructor** can be called from a type. Actually this is how it has to be used to get back a properly created instance.
- Class methods can be defined as **class procedure** (or **function**) Name; or **class procedure** Name; **static**;, but not with the **static** keyword alone. It resolves the issue mentioned in 4.3, i.e. how **static** methods can behave unexpectedly. This case here simply cannot occur.
- Both versions of class methods can be called from a type (remember in case of **object** simple c**lass procedure** was not callable).
- Parent type class instances can be created with the descendant's instance constructor, but not vice-versa.

A long example showing what is possible and what is not:

```
type
  tMyClass = class
    IInstance : integer;
    IStatic : integer; static;
    class var IClass : integer;

    class constructor CClass;
    constructor CInstance;

    procedure PInstance;
    // Unlike object the next line is not allowed
//  procedure PStatic; static;
    class procedure PClass;
    class procedure PClassStatic; static;

    end;

class constructor tMyClass.CClass;
  begin
  writeln('tMyClass.CClass called');
  IStatic          := 112;
  IClass           := 113;
  end;

constructor tMyClass.CInstance;
  begin
  writeln('tMyClass.CInstance called');
  IInstance        := 121;
  IStatic          := 122;
  IClass           := 123;
  end;

procedure tMyClass.PInstance;
  begin
  writeln('tMyClass.PInstance - ',IInstance, ' ' , IStatic, ' ' , IClass, ' ' ,
ClassName);
  writeln('tMyClass.PInstance @ ',Self.IInstance, ' ' , Self.IStatic, ' ' ,
Self.IClass, ' ' , Self.ClassName);
  end;

class procedure tMyClass.PClass;
  begin
  writeln('tMyClass.PClass - '(* , IInstance, ' ' *), IStatic, ' ' , IClass, '
' , ClassName);
  writeln('tMyClass.PClass @ '(* , Self.IInstance, ' ' *), Self.IStatic, ' ' ,
Self.IClass, ' ' , Self.ClassName);
  end;

class procedure tMyClass.PClassStatic;
  begin
  writeln('tMyClass.PClassStatic - '(* , IInstance, ' ' *), IStatic, ' ' ,
IClass, ' ' , ClassName);
//writeln('tMyClass.PClassStatic @ ', Self.IInstance, ' ', Self.IStatic, ' ' ,
Self.IClass, ' ' , Self.ClassName);
  end;

type
  tMyClass2 = class(tMyClass)
    class constructor CClass;
    constructor CInstance;

    procedure PInstance;
    class procedure PClass;
    class procedure PClassStatic; static;

    end;

class constructor tMyClass2.CClass;
  begin
  writeln('tMyClass2.CClass called');
  IStatic          := 212;
  IClass           := 213;
  end;

constructor tMyClass2.CInstance;
  begin
  writeln('tMyClass2.CInstance called');
  IInstance        := 221;
  IStatic          := 222;
  IClass           := 223;
  end;

procedure tMyClass2.PInstance;
  begin
  writeln('tMyClass2.PInstance - ',IInstance, ' ' , IStatic, ' ' , IClass, '
' , ClassName);
  writeln('tMyClass2.PInstance @ ',Self.IInstance, ' ' , Self.IStatic, ' ' ,
Self.IClass, ' ' , Self.ClassName);
  end;

class procedure tMyClass2.PClass;
  begin
  writeln('tMyClass2.PClass - '(* , IInstance, ' ' *), IStatic, ' ' , IClass, '
' , ClassName);
  writeln('tMyClass2.PClass @ '(* , Self.IInstance, ' ' *), Self.IStatic, ' ' ,
Self.IClass, ' ' , Self.ClassName);
  end;

class procedure tMyClass2.PClassStatic;
  begin
  writeln('tMyClass2.PClassStatic - '(* , IInstance, ' ' *), IStatic, ' ' ,
IClass, ' ' , ClassName);
//writeln('tMyClass2.PClassStatic @ ', Self.IInstance, ' ', Self.IStatic, ' ' ,
Self.IClass, ' ' , Self.ClassName);
  end;

var
  MyClassAsMyClass   : tMyClass;
  MyClassAsMyClass2  : tMyClass;
  MyClass2AsMyClass  : tMyClass2;
  MyClass2AsMyClass2 : tMyClass2;

  begin
  writeln('tMyClass');
  // Instance constructor can (and shall) be called from type
  tMyClass.CInstance;
  // Class procedure can be called from a type
  tMyClass.PClass;
  // Class and static procedure can be called with no Self
  tMyClass.PClassStatic;
  writeln;

  writeln('tMyClass2');
  tMyClass2.CInstance;
  tMyClass2.PClass;
  tMyClass2.PClassStatic;
  writeln;

  writeln('MyClass as MyClass');
  MyClassAsMyClass := tMyClass.CInstance;
  with MyClassAsMyClass do
    begin
    PInstance;
    PClass;
    PClassStatic;
    end;
  writeln;

  writeln('MyClass as MyClass2');
  MyClassAsMyClass2 := tMyClass2.CInstance;
  with MyClassAsMyClass2 do
    begin
    PInstance;
    PClass;
    PClassStatic;
    end;
  writeln;

  // Descendant cannot be created with the parent constructor
//MyClass2AsMyClass := tMyClass.CInstance;

  writeln('MyClass2 as MyClass2');
  MyClass2AsMyClass2 := tMyClass2.CInstance;
  with MyClass2AsMyClass2 do
    begin
    PInstance;
    PClass;
    PClassStatic;
    end;
  end.
```

In this example no **destructor** is called. Please do not make this mistake in real life!

6.9 Cross references

In case of advanced records we covered the topic, that because records are variables, two record types cannot contain vice versa a field with the other type.
In case of **object** types it can be done if we use pointers rather than variables.
In case of **class** types it is even easier, since even a seemingly **class** type variable is actually a **class** type reference. So the following code is perfectly OK:

```
type
  tWife = class; // a forward declaration
  tHusband = class
    Name : string;
    Spouse : tWife;
    end;
  tWife = class
    Name : string;
    Spouse : tHusband;
    end;
```

6.10 Visibility

Class type can have all the visibility sections mentioned for **object**s, and there is a new one: **published**: https://www.freepascal.org/docs-html/ref/refsu37.html.
Fields defined in the **published** section are visible as **public**, but has some extra features useful in limited cases. First the fields put in the **published** section must be of **class** type and cannot include **array** properties. If the compiler is not in {$M+} mode, **published** is equivalent to **public**, but in M+ mode, it generates a Run-Time Type Information (RTTI) for the methods and properties in the **published** section.

6.11 Message methods

In **class** another new method type is introduced, called **message**: https://www.freepascal.org/docs-html/ref/refsu31.html. This is basically an extra identification of the method, a method that can be called directly (not recommended) or through a special TObject method called Dispatch. The main purpose of **message** is however not this, but to provide an easy interface for various callback functions that do not necessarily know the name of the method.

7 TObject

So far the keywords used cover language elements. **record** and **object** are two keywords that create an empty object (in the everyday meaning), what can be decorated with fields, methods, properties, etc.
As mentioned above **class** is a bit more advanced. While it is also a language element that in itself define a lot of properties of the object created with it, like pointer dereferencing, visibility levels supported, etc., it is also a half ready creature in itself.
Every time a **class** is defined, it is based on one or more existing class types. If we do not specify it, it automatically uses TObject as the parent class type.

It is a bit of magic to me, that TObject is defined as a **class** without a parent, and a **class** without a parent is automatically uses TObject, still TObject does not end up in an infinite definition loop, but this is how it is.

It is a bit of inconsistency in my mind that the root element of all classes is called TObject and not TClass, but again, this is how it is, **object** has no root element, so TObject was free to use for a **class**. This also added the opportunity to use TClass again for something else, other than the root element of all **class**es.

In this chapter I try to cover some aspects of TObject not touched before, but for full details sometimes the best is to look at the definition or even the full source code of TObject.

7.1 Create and Destroy

I mentioned above the TObject has a predefined **constructor** and **destructor**, called Create and Destroy.
It is customary to redefine the default **constructor** even with parameters if needed. A **constructor** is never **virtual** and often the **inherited** Create is not called. This is not a problem, the compiler takes care with the **constructor** keyword to call all the necessary methods to reserve the memory.
Destruction of a class instance is different according to the documentation https://www.freepascal.org/docs-html/ref/refse39.html. It says that the **destructor** must have the name Destroy, while in my tests this is not the case. But according to the documentation only Destroy calls (when called the inherited one explicitly) FreeInstance needed to free up the memory.

7.2 Message handling

The handling of the **message** keyword and the mechanism behind require some special methods. These are all specified in TObject. Please, look at Dispatch, and DefaultHandler.

7.3 Other features

TObject defines some more methods that can be useful
- UnitName is a function that as its name shows returns the name of the **unit** the **class** is defined.
- Equals in an instance level comparison function.
- ToString to make the class somehow printable. In the default version it returns the class name.

8 TClass

To continue the confusing things, as TObject is not a root element to an **object** hierarchy, TClass is not a root element for **class** either (since that is TObject).

TClass is a simple reference to a new language element:

```
type
  TClass = class of TObject;
```

In a TClass type one can store a type of a class, i.e. a descendant of TObject.

As we saw above, instances of TObject and descendant can be initialized with their own or with a descendant type's **constructor**. Nonetheless the variable always has a class that can be checked through the ClassName property.

While the class instance is not created the variable refers to the class type used when the variable was declared.

Once the class instance is created, we might get a strange result, as it can be seen in the above test program for MyClassAsMyClass2. It is a tMyClass type variable, but as it is created with the constructor of tMyClass2, it more or less thinks that it is of type tMyClass2. When we print the ClassName property or call the ToString function, we get tMyClass2 as a result. However, when we call the PClassStatic **procedure** the result is tMyClass. It is because it has no **Self** parameter and the hard coded type to use when the **static class procedure** is called, it is tMyClass.

Anyway, the important thing is that a **class** type variable cannot hold different types of uncreated TObject descendants. For that a new language structure, "class of TObject" can be used. TClass is simply a short type declaration for class of TObject.

Since a TClass can hold the type of a **class**, not the class instance itself, the allocation is not done directly from the class instance, but through special methods that return the class type. Those are ClassType and ClassParent.

For any class instance being declared as a type that is descendant of TObject, the following code would discover the whole object hierarchy:

```
procedure Discover(AnObject: TObject);
  var
    Reference: TClass;
  begin
  Reference := AnObject.ClassType;
  while Reference <> nil do
    begin
    Writeln(Reference.ClassName);
    Reference := Reference.ClassParent;
    end;
  end;
```

Please note, that in case of MyClassAsMyClass2, the discovered hierarchy starts from tMyClass2 as the instance is created with its **constructor**, regardless that it was declared as tMyClass1.

A TClass variable can also get a type directly:

```
type
  tMyClass = class
    end;
var
  Reference : tClass;
begin
Reference := tMyClass;
writeln(Reference.ClassName);
end.
```

9 Unit as an object

Units are essential parts of any structured program, but they are not really seen as an OOP structure to use.
In this chapter I try to show, why I often use **unit** as a replacement of the previously listed OOP structures.

Two things to note. It is to be used when the "object" created as a **unit** will have one and only one instance and there is no inheritance involved.
The first comes from the fact that the **unit** or actually the variables therein are created when the program starts and no multiple copies can be made, no variable can be declared as of a unit type.

The second is a bit more tricky. Units referencing each other is common practice and as such, some sort of inheritance can be understood in this context, but I think the complexity in this case does not justify the usage instead of a "real" OOP programming element.

9.1 When to use?

I use a **unit** as an OOP replacement, when the following conditions are met:
- It is a large subsystem of my program
- It has many variables
- It has to be reached from many different parts of my program.

My best example is a log recording **unit**.
- It is a clear thing of its own, i.e. can be considered an "object"
- It has many variables, like files to use, e-mail addresses to send alerts to, etc., so simple methods are not sufficient.
- At the beginning of the program I would create one and only one copy of it.
- It is used from many places in the program.
- It is used frequently, so it is not economic to use always the VariableName.MethodName approach, but because the call to the object is typically one line at a given point, it is not logic to use with VariableName do MethodName either.

In the next points I check how the various elements used in OOP can be achieved with a **unit** as an object.

9.2 The unit as an object

When a program is started, all its **unit**s are initialized and all the global variables therein are placed in the memory.

From this point of view we can think of the **unit** as a **record** or **object** that is immediately created. The fields of the object are the global variables of the **unit**.

Elements of a **record** or **object** we refer as VariableName.ElementName and this can perfectly be done here as well: UnitName.ElementName.

9.3 Visibility

As we saw earlier, advanced **record**s that do not handle inheritance, can define two basic types of visibility, **private** and **public**.
These two can almost identically be reproduced, by putting a certain element, let it be **var**, **const**, **type** into the **interface** or **implementation** section of a **unit**. Actually, as shown in the next point, it is even a bit better.
The third type of visibility used in **record**s, **strict private** has no real meaning here, as the "object" and the **unit** are the same in this case.
Similarly, other visibility levels, like (**strict**) **protected** and **published** cannot be interpreted here.

9.4 Fields, Types and Consts

Fields of a **unit** are the variables declared within. Since **unit** is a very high level container, any type of variable can be declared.

Elements defined as **object** or **class** can also include so called nested **type** and **const** declarations. This is typically done to limit the visibility of any such declaration and to avoid name conflicts.

Actually this trick really works to avoid name conflicts, but it is not true that it limits visibility. Consider the following code:

```
type
  tC = class
    strict private
      const Number : integer = 123;
    private
      type tT = record
        Field : integer;
        end;
    end;
var
  C : tC;
  T : tC.tT;
  I : integer;
begin
I := tC.Number;
T.Field := 456;
Writeln(I, ' ', T.Field);
end.
```

It will run happily, not even a hint or warning that we are using **private** type and a **strict private** constant outside the **class**. From the point of view of **class**, it is logic and not a bug:
For tC.Number we actually use the Number as a tC field and tC can see its **strict private** declaration.
The T : tC:tT declaration is a bit more questionable why it works, but as we are in the same **unit**, **private** is OK to work. If we change it to **strict private** it will not compile.

If we use **unit** as our object, the visibility is regulated by where we place it. If we put a constant in the **implementation** section, it is really **private**, not available even as UnitName.ConstantName, so we can say that a **unit** as an object gives better privacy of declarations than a **class** usage.

9.5 Properties

Read only properties can easily be reproduced by a **function**, but I am not aware of a trick to make also a Write property.

9.6 Constructor, destructor

Since the **unit** as an object is most similar to an **object**, with class elements only it does not need a Create like **constructor**. What it more needed is an Initialization, just remember back the discussion on the names of an **object** and a **class constructor**.

The good new that initialization and finalization of objects of **unit** type is solved.

Most often, a **unit** just ends with a **end.**
However if we use both **begin** and **end**., then anything written in between is run before the program starts, i.e. it behaves the same as a class **constructor**. We can replace the **begin** with **initialization**, with the same behavior.
If we also want a **destructor** like part to finalize our object, we can use three keywords: **initialization**, **finalization**, **end.**. The code between **initialization** and **finalization** is run when the program starts and the code between **finalization** and **end.** runs when the program terminates.

9.7 Namespace resolution

One thing why I particularly like to use **unit** as an object type, is that the **unit** name as a namespace is automatically resolved. Basically with the **uses** keyword, we automatically and hidden include a huge **with**, for the referenced **unit** and so, we never need to use it again.

In some rare cases, for incorrectly chosen element names, there might be name conflicts. If a name appears in multiple **unit**s then the order of the **unit** names in the **uses** will decide which element is used. However if we have this problem and want to use a different one, we can still return to the UnitName.ElementName version, what resolves the name conflict.

For the very advanced problem of the dotted **unit** name and their namespace resolution, please read https://www.freepascal.org/docs-html/ref/refse113.html.