Advanced interactive interfaces with Access

Building Interactive Interfaces with VBA

—

Alessandro Grimaldi

Apress®

Advanced interactive interfaces with Access

Building Interactive Interfaces with VBA

Alessandro Grimaldi

Apress®

Advanced interactive interfaces with Access: Building Interactive Interfaces with VBA

Alessandro Grimaldi
Frankfurt am Main, Germany

ISBN-13 (pbk): 979-8-8688-0807-4 ISBN-13 (electronic): 979-8-8688-0808-1
https://doi.org/10.1007/979-8-8688-0808-1

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Laura Berendson
Editorial Assistant: Kripa Joseph

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

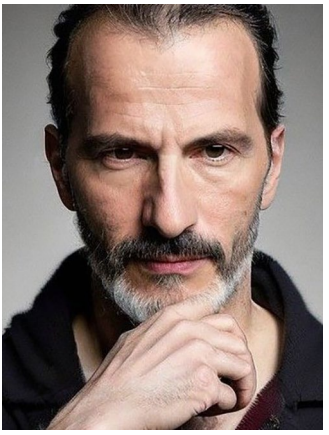
Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

To my parents. Nothing would have been possible without them.

Table of Contents

About the Author



Alessandro Grimaldi was born in Rome, Italy, where he first approached the computer world in 1982. He has been a professional VBA developer since 1998. For several years, Alessandro consulted for the World Food Programme (WFP), a major United Nations agency for which he worked in Afghanistan, North Korea, Ethiopia, and Italy. He has also worked in Vienna, Austria, for the CTBTO, another UN agency. Since 2014, he has lived in Frankfurt, Germany, where he worked for the European Central Bank for about five years. In all these places, he has developed VBA tools, ranging from simple automation tools to complex, multiuser, distributed, enterprise-level applications.

In recent years, Alessandro has produced several videos about drag and drop and the scrolling timeline and delivered live workshops and presentations explaining these techniques. He has an online VBA shop (AlessandroGrimaldi.com/Shop) where he sells VBA tutorials, workshops, and tools. He also has a YouTube channel (@AlxGrim) where he publishes VBA-related videos.

About the Technical Reviewers

Simone Bardi, a database solution designer and developer, has over 20 years of national and international experience in developing tools for data collection, monitoring, and analysis for organizations such as the United Nations World Food Programme (Afghanistan and Italy), UN International Organization for Migration, European Central Bank (Germany), and Vodafone Italy, among others.

Simone has a strong background in MS Access databases (VBA) and MS Excel applications (VBA) for data collection and analysis as well as in cyber security, with many years of experience in ICT security in the private telecommunications sector, focusing on GDPR-related matters, data protection, and fraud prevention.

Colin Riddington has been an Access developer for over 25 years, with many years creating database solutions for schools in the United Kingdom. After taking early retirement from teaching, he set up his own company, Mendip Data Systems, to focus on database development and consultancy. He has been awarded Microsoft Most Valuable Professional (MVP) status for the past three years, starting in 2022. Colin is co-president of the Access Europe user group and a member of the Access Forever team. He is also active on many forums with the username Isladogs. Colin's Isladogs on Access website includes many Access articles, example apps, sample code, and security challenges, together with commercial applications for businesses, schools, and developers. He also runs his own YouTube channel. His focus is on stretching the boundaries of what can be achieved using Access.

Acknowledgments

There are several people I have to thank, who somehow contributed not only to this book but to my work in general.

Stefano Valenzi[1] has been a brother to me for the last 30 years, always helping me and supporting me with his immense knowledge and experience. He gave me the idea to write this book.

Simone Bardi[2] is a good friend of mine. His VBA and logical skills helped me a lot in developing my initial interfaces, and I still call him whenever I need a second brain. He's one of the reviewers of this book.

Karl Donaubauer (Access MVP),[3] the "great guru" of the Access community worldwide. In 2021, he invited me to talk about my work at his Access Developer Conference that year, and since then, my career has taken a decisive turn.

Colin Riddington (Access MVP),[4] the second reviewer of this book. I consider him my mentor and my guide in the complex world of online meetings. Extremely supportive toward me, he did me the honor of nominating me cochairman of AUG Europe.

David Nealey[5] is a fan of my work, and I do admire his. With almost no line of code, he can create amazing infocharts in his Access applications. He's always been very supportive toward me, even choosing me as his cochairman for his LinkedIn group "Modern Access Design."

[1] www.linkedin.com/in/StefanoValenzi/

[2] www.linkedin.com/in/SimoneBardi/

[3] www.donkarl.com/

[4] https://isladogs.co.uk/

[5] www.linkedin.com/in/DNealey/

Introduction

In April 2021, I was invited to the Access Developer Conference (DevCon), probably the biggest and most important gathering of Access developers from all around the world (well, mainly the United States and Europe). Karl Donaubauer (25 times MVP, at the moment I'm writing), the organizer, had happened to see some of my Access works on the Web and asked me to show them at the convention. Despite my previous decades of activity, that was my first exposure to the international Access community, and I didn't really know what to expect. But it went very well; my work received a lot of compliments, and no one in the audience mentioned they had already seen anything similar. Apparently, I had created something new! During the convention, I met David Nealey, a talented Access developer who creates amazing charts in Access practically without code, something that has always amazed me. He forged the term "Access on steroids" to highlight the originality and the visual impact of my interfaces.

I am somehow the opposite of David: I write tons of code, and sometimes I write code to do operations that Access does by default just because I like to have control on every single bit (pun intended) of my applications. And I like to experiment. In the early 2000s, I had this weird idea of experimenting on interactive interfaces with Access, and eventually, the first "drag-and-drop" engine was born. Since then, I have applied this basic engine to almost all of the applications I've been developing through the years, with several variations and improvements, always experimenting, always trying to go "beyond," always pushing Access to its limits.

Why this book? During the last few years, I produced several tutorials and workshops (check my YouTube channel and my online VBA shop), but I eventually decided to collect everything in one single place to reduce the fragmentation and explain things in a more coherent way. I tried to make things as clear as possible, to reach not only the experienced developers but also those with a more limited experience who want to dive deeper into this amazing programming language – even though, of course, a full and solid knowledge of VBA is absolutely required, as the goal of this book is NOT teaching VBA.

We'll start with a quick overview of my favorite "best practices" and a short digression on VBA classes. And that's because in the third part, when I discuss some of my favorite dynamic interfaces, I will make large use of those very personal conventions, which (I'm aware) don't match the more common, "mainstream" conventions. So you'll know what to expect.

I said *dynamic* interfaces: that's how I like to call what I do, more than *graphical* interfaces, because at the end of the day, every interface is graphical, but mine does something more, since there's a lot of things moving, and they allow the user to interact with the objects on the screen, pretty much like Windows does.

One final note: I'm Italian. I studied English mostly by myself, and I'm sadly aware that my English is quite far from being perfect. This means that in this book, you'll find mistakes, wrong words, and expressions that may sound strange to an English native speaker. I refuse to use ChatGPT or any other AI tools to correct it because I still like to use my own brain and I'm not yet ready to turn it off. I just beg you to be so kind as to bear with it – but if you find something really horrible and incomprehensible, please let me know, and I'll be happy to correct it!

Well, I hope you may find at least some interesting input in these pages, and I kindly ask you to mention my name, should you implement in your applications any of the techniques explained here.

CHAPTER 1

Writing Code: Good Practices and Tips

…where I explain my coding conventions, which will be used throughout the entire book.

There is an old saying that goes: "Write your code as if your replacement is a psycho killer." The meaning should be clear: your code should be comprehensible, well written, and easy to understand and modify. Whoever is given the task to read and edit it, you won't want him to get angry with you because you left him with an incomprehensible bunch of messy lines.

Of course, this is slippery ground. We all have a different mind and, above all, a different concept of "order." What is perfectly clear to me may be totally absurd for another programmer. That is why what follows is by no means a list of "best" practices and tips: it's just a list of "good" practices and tips that have been working well for me in the last decades. Depending on your experience, some *are* obvious, some *may* be obvious, and some may be *less* obvious. In any case, I'm not claiming you *must* embed them in your programming style: all I'm saying is that they proved to be reasonable habits, and I hope that some of them may make some sense to some of you. In any case, I'm going to use *these* conventions in the rest of the book, as I'm going to show you *my* original code.

A. Grimaldi, *Advanced interactive interfaces with Access*,
https://doi.org/10.1007/979-8-8688-0808-1_1

1.1. Option Explicit

I always use the `Option Explicit` clause at the top of every module. *Always*. This forces me to declare (`Dim`) every single variable I use in my code, allowing VBA to check if the values I assign them are compatible with their declared type. This can dramatically reduce some kinds of errors – for example, syntax errors. Consider the following fragment of code:

```
Public Sub Multiply(num1, num2)
   Debug.print numl * num2
End Sub
```

Of course, this is very basic, but it's just to illustrate the point. If you call the function (e.g., from the Immediate Window), you will always get a result of 0. Can you understand why? If you look closer, you will notice that the first parameter of the function is <num1> (N-U-M-**1**), while inside the function, it's misspelled as <numl> (N-U-M-**L**). Finding these kinds of errors may be rather difficult and time-consuming, particularly if the misspelled variable is a global one and appears several times throughout your code.

What happens if we use `Option Explicit`? When we run the function (or when we compile the code), the execution stops, the misspelled variable is highlighted, and an error message box says "Variable not defined."

There are even subtler errors that may occur due, for example, to a variable name used twice in different contexts, maybe holding different value types, etc. The list of errors that can be avoided using `Option Explicit` is too long to mention: I just use it every time and strongly suggest you do the same.

1.2. Option Compare

Strangely enough, many VBA programmers disregard this statement, which automatically appears as the very first line of each new code module. Yet, it's an important statement that can dramatically change the behavior of our code, so it's a good thing to thoroughly understand how it works. It basically has to do with string comparisons, for example, in `If` statements. Its parameter has three possible values:

`Option Compare Database`

This option is only available in Access (not in Excel, Word, etc.). Practically speaking, the string comparison is based on the sort order determined by the local international settings, so it follows the rules of the local language.

`Option Compare Text`

The comparison is performed on a case-insensitive basis, so "`John`" and "`john`" are considered the same string.

`Option Compare Binary (default)`

The comparison is based on the binary representation of the characters – practically, their ASCII codes. In this context, "`John`" and "`john`" are different.

An additional note of the term "default," which in this case can be a little confusing. When you create a module, Access adds `Option Compare Database` on the first line, so this is actually a kind of "default." The point is that if you remove that line, and if you don't make explicit which of the three methods you're using, Access assumes it's the `Binary`, which makes it the real "default."

Choosing one or the other of these statements can make a substantial difference. The results of your code can be greatly impacted, so you should pay some attention to this choice. For example, if your code

checks for duplicates, you'll probably have something like `if (newName = oldName) then...`. This kind of instruction is influenced by the `Option Compare` directive, and you may or may not want to consider the different letter cases.

A real-life example (it happened to me!): I was checking the field names of a recordset, and something kept going wrong. I finally realized that the test `fld.Name = "Title"` kept failing because the actual field name was `title` and there was no `Option Compare` directive specified – so `Option Compare Binary` was active. Lesson learned.

1.3. Variable Names

Sometimes we tend to be lazy and name variables with short, meaningless names: `tID`, `clNo`, `pAvg`, and such. This is NOT a good practice. Even though they can be extremely clear in your mind, it makes the code hard to read, hard to understand, and almost impossible for anyone to follow your procedures. And this includes YOU if you happen to look back again to your code after a few months.

Spend some more time, but give your variables comprehensible names: `tempID`, `clientNumber`, `priceAverage`, and so on. It may be OK to use a name like `i` when you run a loop:

```
For i = 1 to 10
```

It's a rather standard practice, and no one will blame you for that. Of course, a more meaningful name is always well accepted, for example:

```
For clientNdx = 1 to 10
```

is definitely better.

Talking about variable naming, there are two vexed questions:

1. *What case should I use?*

   Basically, there are four styles which are widely used:

   Snake Case: number_of_registered_clients

   Letters are all lowercase, and words are separated by an underscore character. It's typically used in Python. A variation is the so-called "Screaming Snake Case," with all capital letters such as in `NUMBER_OF_REGISTERED_CLIENTS`.

   Personally, I do not like it very much, as you need to type the extra character "_" for each word. Typically, I use the "Screaming Snake Case" to declare constants only.

   Kebab case: `number-of-registered-clients`

   Like the previous one, it just uses a dash instead of an underscore. You see a lot of that in web addresses. It goes without saying that I do not like it either because of the extra characters. Anyway, it's not accepted in VBA.

   Pascal case: `NumberOfRegisteredClients`

   No additional characters, here: every word is capitalized, so it's easy to see where a word ends and another begins. This style is widely used, especially for naming classes. And this could be my favorite, if it wasn't for the fourth style…

   Camel Case: `numberOfRegisteredClients`

What's the difference? The first letter, which is in lowercase. This style is especially used in Java and Javascript for creating functions, methods, and variable names. Why do I prefer this? I don't know, really. It's just a personal preference, without a rational explanation.

Whichever style you decide is best for you, the important thing is that you maintain coherence throughout your whole application. Mixing styles may be confusing and definitely gives a sense of disorder, inaccuracy, and superficiality.

2. *Do I have to add a prefix to explicit the variable type?*

   This discussion has always been very popular. If I declare the variable `itemNumber` as an integer, do I have to name it `intItemNumber`? If I declare `clientName` as a string, should I actually name it `strClientName`? Well, in my opinion, the answer is a resounding *no*. If you choose the name wisely, most of the times the prefix is useless. I mean, `clientName` is obviously a string. `isOK` is clearly a Boolean, so do we really need to name it `blnIsOk`? `itemCodeNumber` is a number, intuitively an integer or a long (does it really matter? Do you make calculations on item code numbers?). I think these prefixes reduce the readability, overloading the code with (mostly) useless information.

   Some cases are borderline, I know. For example, `itemCode` is an integer, a long, a string, or what? Well, it's not that important to me. Once I see that `itemCode` is declared as a string, I do not find it too

difficult to remember it when I meet this variable in the code. But again, this is how MY mind works. Yours may very well be very different and appreciate those extra characters in every occurrence of your variables.

1.4. Private/Public

We all know that a variable or a function defined in a code module is always public. And we all know that in a form module, a variable or a function is private by default. But I think it's a very good practice to always make it explicit. So, in a module, I never write

```
Sub somePublicSub()
   [...]
End Sub
```

but I always go for

```
Public Sub somePublicSub()
   [...]
End Sub
```

Similarly, for the private routines, I always write

```
Private Sub somePrivateSub()
   [...]
End Sub
```

I think it makes everything much clearer and unambiguous.

And if I have a lot of subs/functions, sometimes I go one step further and prefix `loc` (for *local*) to all private routines:

```
Private Sub locSomePrivateSub()
   [...]
End Sub
```

This way, in the procedure combo box, they will be all grouped together, and it's easier to tell a public function from a private one, as shown in Figure 1-1.

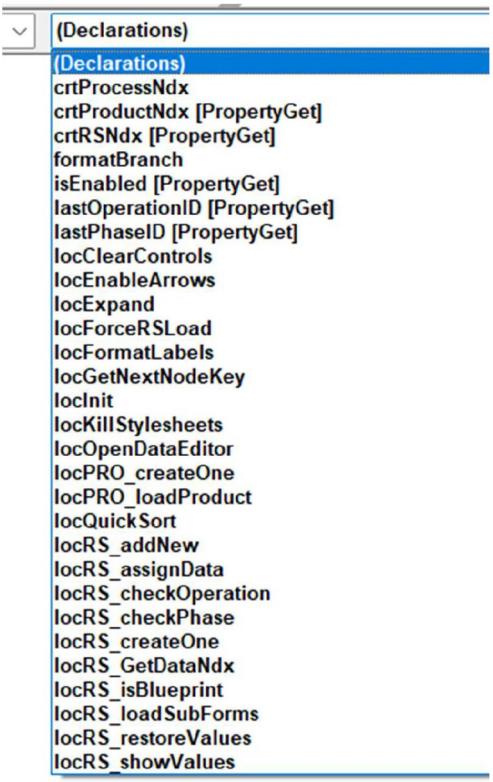


Figure 1-1. *List of procedures*

1.5. User-Defined Types (UDT)

Most of the problems with variable names disappear when you use UDTs. A UDT looks like this:

```
Private Type recClient
   clientName As String
   Code As String
   officeNumber As String
   personalMobileNumber As String
   officeMobileNumber As String
   Date as Long
End Type
Private Client as recClient
```

Why are UDTs so great? For several reasons:

- They allow you to group a coherent set of variables (as many as you need) under one single name (in the previous example, `Client`). You only need to remember *that* name and forget about the many single variables.
- You exploit VBA's Intellisense engine. When you type `Client.` (with a dot) in your code, you can see all the variables inside the structure. You can select one with the keyboard arrows or just type its first letters and then hit TAB.
- For this reason, you can use longer and more meaningful names (see the two telephone numbers in the preceding example) since you do not have to remember them nor to type them.

- A medium project can have dozens of variables. It's easy to forget one or two of them, and you may forget to update them when needed. This can't happen with a UDT since you always have the list of all the variables at a glance.
- In a UDT, you can also use names that normally are forbidden, as they are part of the standard VBA syntax, such as `Date` in the preceding example. And this is possible because since they are enclosed in a higher level structure, there's no space for ambiguity: Access knows that a reference to `Date` and a reference to `Client.Date` are two different things.

Personally, I always use a (public) UDT named `recGlobals` in all my projects, in a module, collecting all the global variables. Then I declare

```
Public Globals as recGlobals.
```

And that immensely helps me keep track of my global variables (which in a big application can be *a lot*), to reduce typos, and to make the code more readable. Besides, in every form module, I always use a similar technique for the local variables, defining

```
Private Type recLocals
   [...]
End Type
Private Locals As recLocals
```

Side note: Why do I prefix `rec` to a UDT name? Historical reasons. In 1987, I had to work with COBOL, and COBOL had this type of structured data, called "record," extremely similar to a UDT in VBA. I used the "rec" prefix a lot, to name those records, and I kept this habit when I moved to VBA.

1.6. Service Functions

A *service* (or *convenience*) *function* is a small function that performs a single, specific task; it doesn't really matter if big (tens of lines) or small (one single line). For example:

- Performs a calculation with numbers or dates
- Builds a string according to a complex pattern
- Reads a value from a table
- Initializes a set of variables
- Enables/disables controls on a form

A service function can be very useful, in case you need it several times. For example, the following function is only one line long, but it's extremely useful to generate a pseudo-random ID with max four digits:

```
Public Function getRandomID() As Long
   getRandomID = Int(Rnd(Timer - Int(Timer)) * 10000)
End Function
```

If you need to generate IDs in your application, remembering and retyping this line several times can be frustrating, boring, and prone to errors. Instead, you can call `getRandomID()` and forget about the details. Moreover, if one day you decide to change the formula that generates the ID, you will only have to change it once, in this function, leaving the rest of the code unaltered, and all generated IDs will keep their consistency.

If you make such functions `Public`, they become in all respects like a native VBA function, which you can use anywhere in your code, or query, or report, or even in a form control. So, don't be shy, use them often, and appreciate their advantages:

- You do not have to write the same stuff over and over.
- Type less, thus reducing typos and errors.

- You do not have to remember long/complex formulas or code snippets.
- Easier code maintenance and modification.
- Easier coherence and consistency control.
- Code readability (a function name vs. complex code).

1.7. Boolean Parameters

Sometimes, a function needs many parameters, where "many" may vary from person to person, but for me, it means three or four. In my mind, if a function needs more parameters than that, something might be wrong, and the code might need to be restructured. Of course, this is not always the case: a function MAY actually need a higher number of parameters.

In many cases, though, it's not real parameters you need to pass but rather True/False values. For example, if you have a function that formats a date into a string, you may want to be very flexible and specify the following options:

- Show/hide the leading zeros (e.g., `01-09-2024` vs. `1-9-2024`)
- Show/hide the year (e.g., `01-09-2024` vs. `01-09`)
- Show/hide the day (e.g., `01-09-2024` vs. `09-2024`)
- Use two digits for the year instead of four (e.g., `24` vs. `2024`)
- Show/hide the time (e.g., `01-09-2024 09:46:23` vs. `01-09-2024`)
- Show/hide the seconds (e.g., `09:46:23` vs. `09:46`)
- Use AM/PM rather than a 24-hour clock (e.g., `07:00:00 PM` vs. `19:00:00`)

And of course, you may think of even more formatting options. Now, normally you would specify a parameter for each of these options. This, of course, would be a terrible choice. The function signature would look like

```
Function formatDate(dt As Date, _
                leadingZeros As Boolean, _
                showYear As Boolean, _
                showDay As Boolean, _
                use4DigitYear As Boolean, _
                hideTime As Boolean, _
                showSeconds As Boolean, _
                useAM_PM_Format As Boolean) As String
End Function
```

plus, other necessary parameters such as the date format (e.g., DD/MM/YY), the separator (colon, space, dash, etc.), and maybe more, which would obviously result in a huge, unacceptable signature.

I saw many applications where the developer solved the problem by creating a class, using these pieces of info as properties. Frankly, to me, this looks like shooting a fly with a cannon. A UDT would be sufficient.

Or, we can use a binary-based technique usually called "bit-field." Since we'll examine this technique later in this book, this can be a good occasion to see a possible application of it. In a nutshell, the bit-field technique considers the binary representation of a number: since each bit can be either 0 or 1, it can be seen as a "presence indicator" for a set of items. Each bit is logically associated with one specific item. If the bit is 0, the value of its associated item is `False`; if the bit is 1, the associated value is `True`. Let's see how this can help in this case. Consider this enumeration:

```
Enum enmDateOptions
   doLEADING_ZEROS = 1
   doSHOW_YEAR = 2
   doSHOW_DAY = 4
```

```
   doTWO_DIGIT_YEAR = 8
   doSHOW_TIME = 16
   doSHOW_SECONDS = 32
   doUSE_AM_PM = 64
End Enum
```

It's my habit to name an enumeration with the `enm` prefix and a two-word name (in this case, `DateOptions`). Each value is then prefixed with the initials of these two words (in this case, `do`). This makes it easier, when I write the code, to remember which enumeration each value belongs to.

Note how each item has a value which is a power of 2. We won't dive into the details of the binary system for now: all you need to understand here is that this set of values guarantees that every combination (sum) of values is unique. For example, the number 98 can only be obtained by

```
doUSE_AM_PM + doSHOW_SECONDS + doSHOW_YEAR
```

and no other combination can give the same value. The new signature for the function will then be

```
Function formatDate(dt As Date, Options As enmDateOptions)
As String
```

which is much more readable than the previous one with eight parameters. How do we call this function? Here's an example:

```
result = formatDate (Date(), doLEADING_ZEROS Or doSHOW_YEAR
Or doSHOW_DAY)
```

The body of the function will check the value of `Options` and take actions accordingly. In our example:

```
Function formatDate(dt As Date, Options As enmDateOptions)
As String
   if (Options And doLEADING_ZEROS) then ...
   if (Options And doSHOW_YEAR) then ...
```

```
    if (Options And doSHOW_DAY) then ...
    if ...
End Function
```

According to the binary math, the AND operation returns 0 if `Options` does not contain the tested value, otherwise returns a nonzero value (more precisely, the value itself). So, in our example:

```
Options And doLEADING_ZEROS = doLEADING_ZEROS (≠0)
Options And doSHOW_YEAR = doSHOW_YEAR (≠0)
Options And doSHOW_DAY = doSHOW_DAY (≠0)
```

but:

```
Options And doTWO_DIGIT_YEAR = 0
Options And do_SHOW_TIME = 0
```

and so on, so the associated `if` actions won't be executed.

And the best thing is that every time an enumeration value is required, the Intellisense engine shows a list of all the available values, so you do not have to remember them, as shown in Figure 1-2.

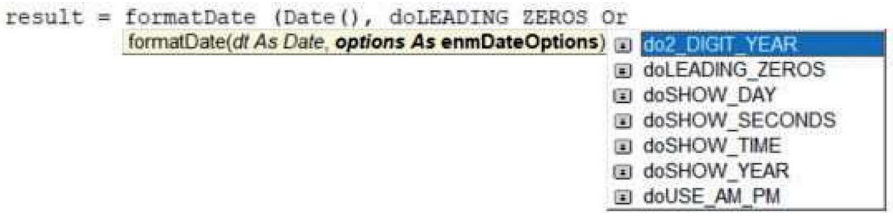


Figure 1-2. *UDT components*

Now, why `Or` and not `+`? In this case, the two symbols are equivalent. But I always prefer to use `Or` in place of the plus sign when I'm dealing with Boolean values because it's more logical and because, in some cases, they produce different results. We'll talk in more detail about this topic in the chapter dedicated to this technique.

Its advantages should nonetheless be already clear:

- Compact signatures.
- Code readability in function calls (meaningful constant names rather than a long series of TRUE/FALSE).
- Scalability and flexibility: You can add enumeration values, but the signature and all the calls to the function remain the same because the number of parameters (two in this case) doesn't change.

1.8. Optional Parameters

Do you ever use the `OPTIONAL` clause in a function signature? Sometimes it's very handy, especially when you find out you need an extra parameter in the middle of your development and do not want to break the existing code. You add the parameter as `OPTIONAL`, and all the code will keep working. Or, more commonly, you do not want to specify the same parameter over and over every time you call that function, so you declare it as `OPTIONAL` giving it that value as default.

```
Function someName(parm1 As Integer, Optional parm2 As
Integer = 0)
```

Personally, I tend to avoid this clause, though. The problem is that when you compile the code, you won't catch all those calls that do NOT specify the optional parameter. So, it's very much possible that sooner or later, you call the function without specifying the optional parameter even when you should and never be aware of that. The output may be wrong, but you may never notice that, and keep thinking that everything is fine – until someone realizes that there's a problem, and this is never a good thing. I'm very careful when it comes to using `OPTIONAL`, and usually, I prefer to take some more time to call the function with all its parameters, every time.