Excel 2016 VBA programming

Visual Basic for Applications

dr Peter J. Scharpff RI

bookboon
The eBook company

DR PETER J. SCHARPFF RI

EXCEL 2016 VBA PROGRAMMING

VISUAL BASIC FOR APPLICATIONS

Download free eBooks at bookboon.com

Excel 2016 VBA programming: Visual Basic for Applications
1st edition
© 2018 Dr Peter J. Scharpff RI & bookboon.com
ISBN 978-87-403-2025-1

CONTENTS

	About the author	**7**
1	**Introduction**	**8**
2	**Development environment**	**10**
2.1	Introduction	10
2.2	Developer tab	11
3	**Macros**	**13**
3.1	Introduction	13
3.2	Recording a macro	13
3.3	Running a macro	17
3.4	Relative Reference	18
3.5	Exercise	22

CMO INSPIRED CONFERENCE
25 OCTOBER | DE VERE BEAUMONT ESTATE | OLD WINDSOR UK
Join Over 100 Chief Marketing Officers & Digital Innovators
Click on the ad to read more

4	**Programming**	**24**
4.1	Introduction	24
4.2	Creating Programs	25
4.3	Objects	29
4.4	Program structures	33
4.5	Variables	36
4.6	Comments	37
5	**The VBA Editor**	**39**
5.1	Introduction	39
5.2	What's in the window?	39
5.3	Working windows	42
6	**Generating Code**	**44**
6.1	Introduction	44
6.2	Recording	44
6.3	Editing Code	46
6.4	Forms	49
7	**Overview**	**57**
7.1	Introduction	57
7.2	Hungarian notation	57
7.3	Immediate window	59
7.4	Object Browser	60
7.5	Exercise	61
8	**Programming techniques**	**62**
8.1	Introduction	62
8.2	Functions	62
8.3	Decision structures	66
8.4	Loops	72
8.5	Key capturing	76
8.6	Variables and constants	78
8.7	Error trapping	81
8.8	Exercise	82
9	**Integrating code**	**86**
9.1	Introduction	86
9.2	Start	86
9.3	Add-in	92

Download free eBooks at bookboon.com

10 Case: an invoice model 97
10.1 Introduction 97
10.2 The invoice workbook 97
10.3 The invoice form 98
10.4 Creating the invoice 105
10.5 Security 109
10.6 User Interface 110
10.7 Extra exercise 111

11 Charts in VBA 113
11.1 Introduction 113
11.2 The workbook 113
11.3 Use of the VBA-app 116

Download free eBooks at bookboon.com

ABOUT THE AUTHOR

Scharpff Consultancy is a consulting organization that has been active in the world of automation for more than a quarter of a century. Information, training and guidance on the use of personal automation tools are key focuses. One of the specialties is facilitating (and executing) individual training programs for PC-users. The model to accomplish this has been created in the early '90s from the last century by Scharpff Consultancy in collaboration with various experts. Nowadays, it is used by most training centers and even by mainstream educational institutions. Individual schooling has proven to be a better method for learning computer skills than the classroom-training forms that are still being used often in the training world.

Scharpff Consultancy has a lot of expertise in developing and publishing materials for individual training purposes. This textbook is a product of that focus. It will allow you to obtain the necessary knowledge and skills on your computer completely on your own.

Dr Peter J. Scharpff RI, who is both the founder of this consultancy agency and the author of this workbook, originated from the field of computer-linguistics. His scientific focus was on the interaction between man and machine, especially on the speech and language technical side of it. Besides developing materials for training purposes for office automation, he also produced many publications on a broad range of topics such as: digital security, privacy, hardware, programming, web design, teleworking, social networks and many more.

Download free eBooks at bookboon.com

1 INTRODUCTION

This workbook is written for the Excel 2016 user who wants to create custom applications, modify recorded macros, or add extra functionality in the spreadsheet program. Macros or applications can be designed to meet specific wishes when you experience that the default possibilities offered by the software are insufficient for your needs. In order to do this, you must have knowledge of Visual Basic for Applications (VBA), and you'll have to develop programming capabilities to a certain degree.

The first chapters of this workbook will cover the basics of programming in Visual Basic for Applications including the setup of a VBA-project, and the basic steps of building a program. In later modules, we will discuss some possible applications and practical examples of VBA in Excel.

The book has a modular layout. Each module is set up in the following way:

- Each chapter starts with a short explanation of the topics that will be addressed.
- If possible, each topic will be presented with a (short) exercise. These exercises describe a possible situation or problem, the way to handle or tackle it, and the final outcome.
- At the end of a chapter, there is often an extra exercise to put into practice what you've learned.

Exercises are labelled with a mouse icon next to the text. Please, focus on these instructions, you don't need to perform all actions or examples mentioned in the text. It can be beneficial, however, to experiment with other possibilities suggested in the text.

Even though it is possible to go through each module independently from the others, not all actions and keystrokes will be mentioned because we assume that you already possess certain skills, for instance the basic operations in Excel. In several exercises you'll find only general instructions and you'll have to figure out yourself how to accomplish the task.

Names of keys and buttons are given between square brackets: [and]. In some cases you have to hit two keys at the same time, we will denote this by a plus sign: [Alt]+[F]. When you have to type two keys consecutively, there will be no plus sign: [F5] [2] [Enter].

Download free eBooks at bookboon.com

Titles, texts and options that you see in your window are marked *italic*, for example: the window *Properties* contains several sections and tabs such as *View*, *Security* and others. File names and folders are shown as **bold** text.

We assume that you have access to the software described here, and you can perform standard operations in Excel and Windows. Only where actions are very different from normal use, they will be mentioned in the instructions.

Download free eBooks at bookboon.com

2 DEVELOPMENT ENVIRONMENT

2.1 INTRODUCTION

Visual Basic for Applications (VBA) is a programming language, or rather an environment that enables you to develop your own programs within Office. This language is derived from (and closely related to) the independent programming environment of Visual Basic in which you can create real programs that can run stand-alone within Windows. These programs can be recognized by their extension: **.exe**.

A programming language is made up of instructions for the computer to perform certain tasks. The program code is structured in a specific way and uses several (complex) methods to instruct the computer to execute these actions.

VBA programs can only work within the Office environment, because they depend on the 'parent' program in which they are written. This workbook will explore the possibilities within the Excel program.

By now, you are probably wondering why you would want to program within a spreadsheet application that already offers so many functions and options. Often, the answer is: to automate routine actions. But you could also use macros to do this, so why write a program?

Programs can intelligently execute specific instructions in cases where you'd want to automate an action but not before you've gathered some information from the user. Before executing the instructions you could communicate with the user with questions like: "What is the initial value for the calculation?" "Where can I find the data?" or "What report do you want to print?"

Programs in VBA are created in the form of projects that offer additional functionality to the basic software. They contain instructions that can use existing information, objects and resources such as windows, tables, charts or buttons. The projects are started from the Office environment – in this case from Excel – and can not function as independent programs outside of that environment.

Therefore, VBA has a specific project window: the VBA Editor. This editor can be seen as a toolkit with specific objects, methods and tools, in combination with several property windows and troubleshooting possibilities.

Download free eBooks at bookboon.com

2.2 DEVELOPER TAB

To enable the use of VBA, the editor and other related items, it is required to activate the *Developer* tab in the ribbon. This tab is not shown by default.

Create a new (empty) spreadsheet in Excel.
Activate the 'backstage' view (*File* tab), click *Options*, and go to the *Customize Ribbon* category.

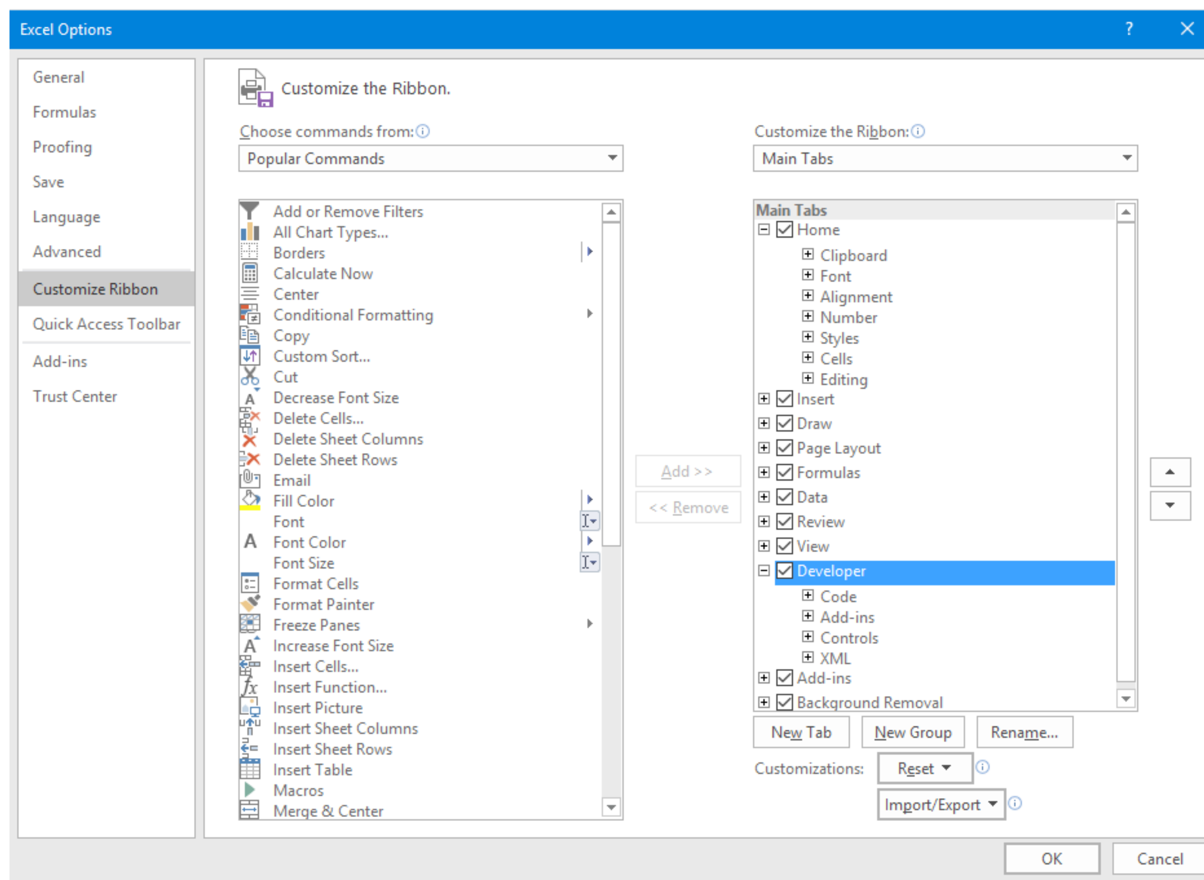


Figure 2.1 Making the Developer tab show

In the right pane with the *Main Tabs* list, click the *Developer* checkbox.
Close the window titled *Excel Options* with [OK].
Click the *Developer* tab in the ribbon of the Excel window and see what options it contains. Read the function or explanation in the tooltips that appear when you hover over each of the elements in the tab.

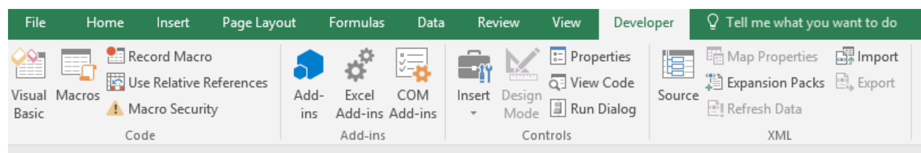


Figure 2.2 Developer tab in Excel

VBA can be used to automate and modify actions to meet specific requirements or wishes. That is in fact also the purpose and process of macros. When you create a macro in Excel to automate routine actions, the corresponding VBA programming code will be generated in the background. You can also enter these instructions yourself; you will then be programming in VBA. We will discuss this further on, let's first take a look at the 'macro'.

3 MACROS

3.1 INTRODUCTION

Macros are small programs, typically intended to perform repetitive actions. They can easily be activated by pressing a key combination that you have linked to a macro. Executing macros lowers the amount of actions – clicks and/or keystrokes – for you to perform a (routine) task.

3.2 RECORDING A MACRO

Macros can be created by having Excel record your actions step by step, as if it were a recording with a tape-, video- or harddisc-recorder (if you can remember what that is). Recording can be started in several ways:

- By clicking the Record Macro button in the *Code* group on the *Developer* tab (see Figure 2.2).
- By using the Macro button in the status bar at the bottom of your window.

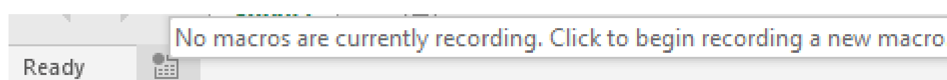


Figure 3.1 Macro button in the status bar

- By choosing the option *Record Macro* from the drop-down selection list of the *Macros* button on the right of the *View* tab. This button is placed there for users that require the macro functionality but have no need for all the possibilities of the *Developer* tab.

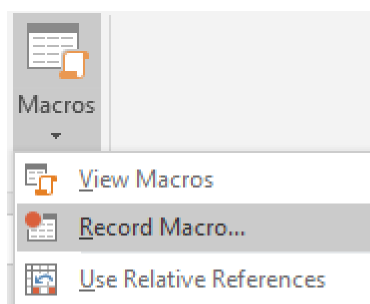


Figure 3.2 Record macro option on the View tab

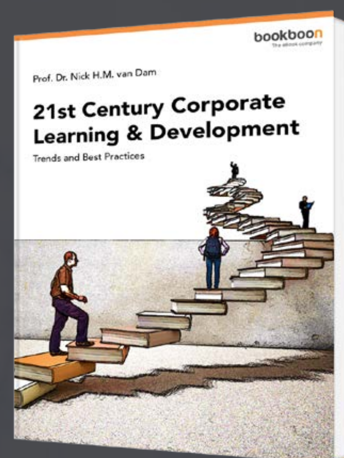
Download free eBooks at bookboon.com

In all cases, Excel will create a special Visual Basic worksheet for the recording of the actions. To illustrate this, we will now record a simple macro that sets the width of a column to a predefined value through the use of a shortkey. This action does not require many mouse clicks or keystrokes, but it is a good example of how a routine action can be replaced by a macro.

Think of a name and a location (workbook) for the macro you are going to create. You can assign a keyboard shortcut to the macro and give it a description. Before you start recording, make sure which actions you need to perform. It might be a good idea to practice the actions first (and undo them) before you start.

Start recording a new macro using one of the methods described above.

Free eBook on
Learning & Development
By the Chief Learning Officer of McKinsey
Download Now



Download free eBooks at bookboon.com
Click on the ad to read more

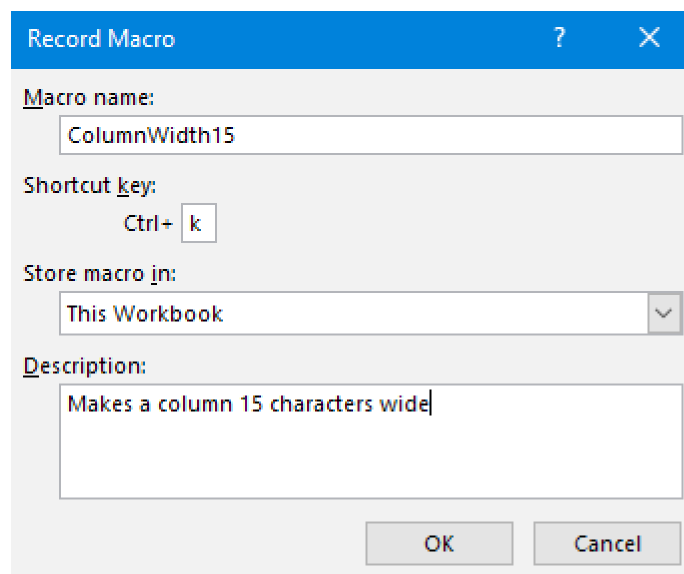


Figure 3.3 Defining a Macro

Complete the window as shown in Figure 3.3 (we have also modified the *Description*) and click [OK].

At the bottom of the screen, next to 'Ready', a button with a block appears. Hovering over this button will show an information box: *A macro is being recorded. Please click to stop recording.* Also, on the *Developer* tab the *Record Macro* button is replaced with *Stop Recording*.

ALL keystrokes or (mouse) actions you perform from this moment on will be recorded in the macro memory until the recording of the macro is stopped. You don't have to rush, take your time to think about what actions to perform.

Now perform the necessary actions to set the width of the current column to 15: e.g. on the *Home* tab, in the *Cells* group, *Format*, *Column Width* (or by using the context menu of a cell in the current column).
Stop the recording of the macro.

The macro is now stored in a special Visual Basic worksheet. Let's have a look at it to see what Excel has registered.

On the *Developer* tab click *Macros* (or on the *View* tab click *Macros*, *View Macros*).

Download free eBooks at bookboon.com

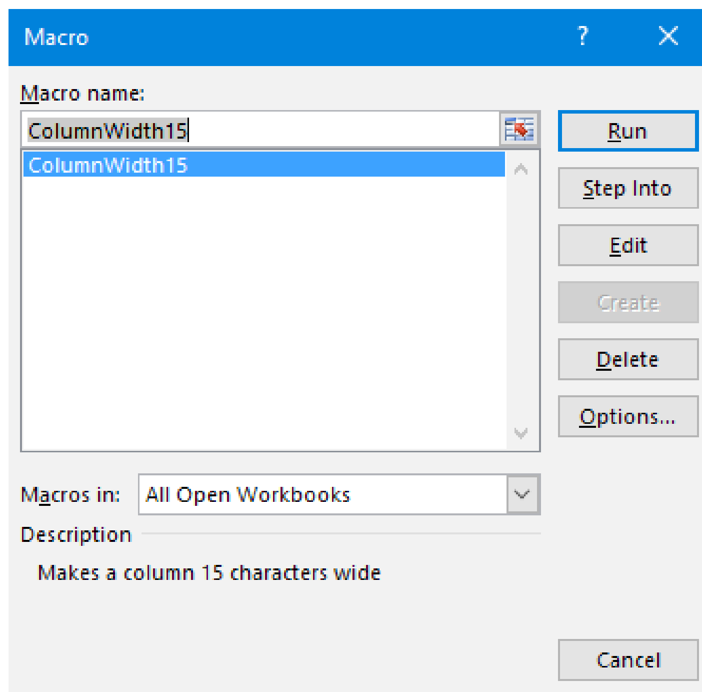


Figure 3.4 The Macro window

This window shows the macro that you just recorded as well as any previously recorded macros. From this list you can run the macro again, for instance to change the size of another column. But now we will have a look at the code behind the macro called *ColumnWidth15*.

Make sure the macro is selected and click the [Edit] button. A new window specifically designed for VBA will pop up and shows the recorded code. The exact details of this code are not relevant right now, we'll get to that later.

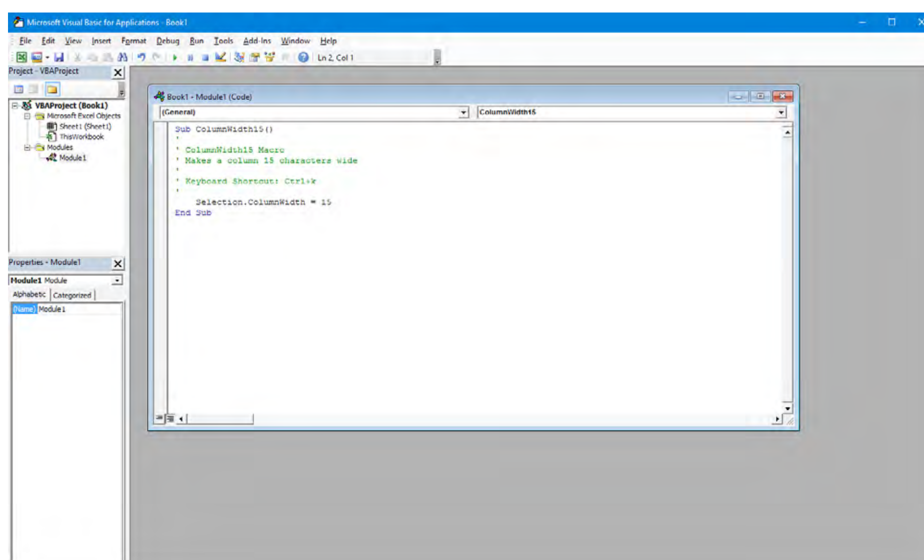


Figure 3.5 VBA Editor

Download free eBooks at bookboon.com

Read the comments (the green texts that start with an apostrophe) and the instruction below them in the window *Module1 (Code).*

This instruction is the only thing actually happening when the macro is executed: set the width of the selected column – the one that contains the active cell – to 15. The comments are there for documentation purposes (for the programmer, and therefore for you), but the lines are ignored entirely by Excel when executing the macro.

3.3 RUNNING A MACRO

You can run the macro you just created again and again, for instance to resize other columns.

Leave the VBA window open (narrowed or minimized if necessary) so you can see it next to the workbook window.
Now select another column in the worksheet and run the macro again (this time using the assigned shortcut).
Select two columns and use the macro to make them 15 characters wide as well.
Select three (not adjacent) columns as a multi-selection and run the macro again.
Make a random column extra wide (by dragging the right border) and use the macro to reduce the width of this column to 15.

So there are several situations where you can use this macro. There is one limitation though, we have stored this macro in the current workbook (see Figure 3.3). This means you can only use the macro when this workbook is open.

Close and save this workbook. Make sure you set the file type to *Excel Macro-Enabled Workbook*, and name the file **Width**(.xlsm). Once your workbook is closed, the window of the VBA Editor will also be closed.

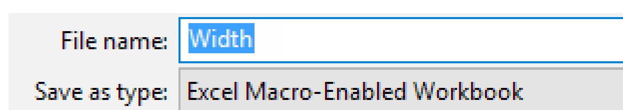


Figure 3.6 Saving a workbook with macros

Create a new workbook (or use an existing one, but not **Width.xlsm**).
Use the keyboard shortcut of the macro [Ctrl]+[k] to adjust the width of one of the columns. What happens?

Download free eBooks at bookboon.com

What does **not** happen is the column width being set to 15, and that's because the macro with the shortcut is not available. The workbook in which the macro is stored, was closed.

What **does** happen is that the window *Hyperlink* appears, because the shortkey [Ctrl]+[k] is by default assigned to *Add a Hyperlink* in the worksheet (hover over the *Hyperlink* button on the *Insert* tab to see the hotkey).

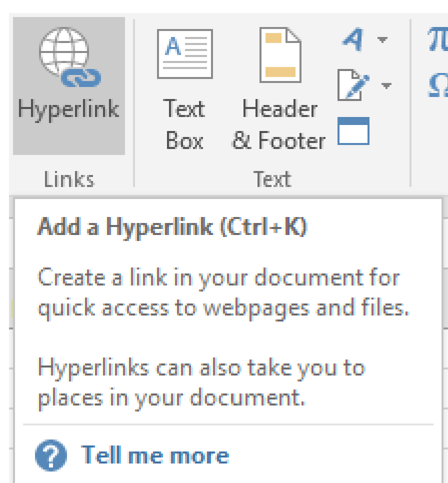


Figure 3.7 Hyperlink button with hotkey

When defining a macro think carefully about the shortcuts you choose. Some may already be in use as default shortcuts or were assigned as hotkeys to other macros! When you link a shortcut to a macro that is available in all workbooks, the original default shortcut will be lost.

Close the *Hyperlink* window, and close the workbook (you don't have to save it).

Running and saving macros is also important when you are going to program in VBA. Many tasks will be started or executed as a macro. We will explain this when the topic arises.

3.4 RELATIVE REFERENCE

The macro that sets the column width works on each column where you run it from. The only thing the macro does when activated, is set the width. This makes the macro 'relative' to the position where the macro is executed: when your cell cursor is in column B, the width of this column is set to 15 characters, when column XY is active the macro will do the same for that column.

But if during recording of the macro actions are required, like navigating to other cells or selecting columns or rows, running the macro again can lead to undesired effects. Excel regards the cell addresses in such actions as absolute references.

Let's create a new macro now that takes into account the relative position of the active cell pointer. For example, suppose you want to put your name and address in three adjacent cells of the same column, anywhere in the worksheet. While typing the name and address you'll need to move the cell pointer down twice. You have to inform Excel that you want to do that relatively to where your cell pointer is.

Start a new workbook.
On the *Developer* tab, in the *Code* group, click the button *Use Relative References* (that option then remains selected).

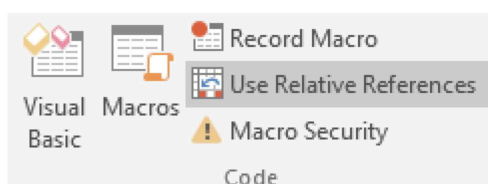


Figure 3.8 Relative recording

Now start recording a new macro named *NameAddress* (stored in *This workbook*, shortcut: [Ctrl]+[n]) that writes your name, address and city in three adjacent cells (like in Figure 3.9 on the left) and then adjusts the column width to the length of the widest cell of the three (AutoFit).
Stop recording.
Select another cell in the sheet and run the macro again to insert your address there.

By having switched on the *Use Relative References* option during the recording, the macro will, if re-run in another location, put the second and third text cell under the first one. Would you have forgotten to activate this option, however, the first text line (Name) would be put in the currently selected cell, but the second line (and third) would end up in the same place where you defined the macro. Let's have a look at how this works.

Via the *Developer* tab, go to the VBA Editor and review the code of this *NameAddress* macro.

Download free eBooks at bookboon.com

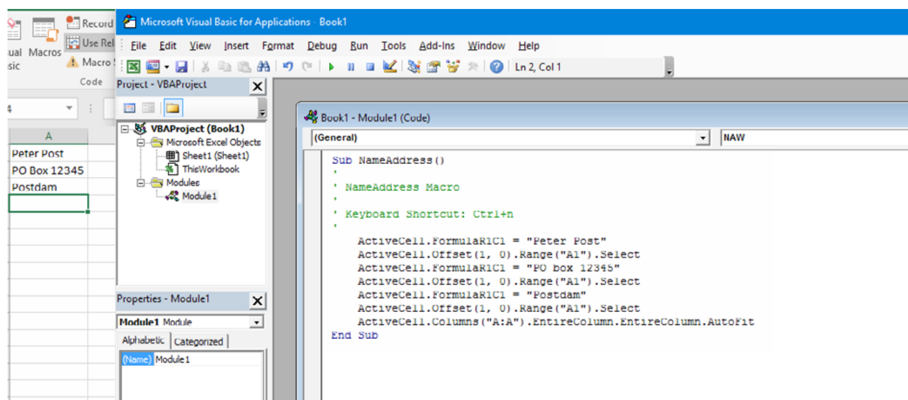


Figure 3.9 Code lines

By the looks of it, you have already programmed quite a few lines. You can now read the instructions line by line and surmise what happens when you run the macro: the program types the Name in the first (active) cell, the cell pointer moves down one line (and zero columns), the macro types the address in the next cell, etcetera. It is important that you learn how to read these lines of code. Once you understand how it works, you could even edit the lines or information. For example, if you'd want to change something in the address, it is easier and faster to edit the code than to redefine the macro.

Having Excel run the macro code step by step is also a possibility. This is a commonly used method for error detection and troubleshooting of the code. We will discuss this extensively later. We will now show you what happens when the instructions in the macro are executed.

Select a cell a bit to the left in the worksheet, position the Editor window in your screen so that you can still see the active cell in the background. Use the *View* menu in the VBA window – there is no ribbon there, you have to use the menus and the toolbars – to show the *Debug* toolbar and place it near the code.

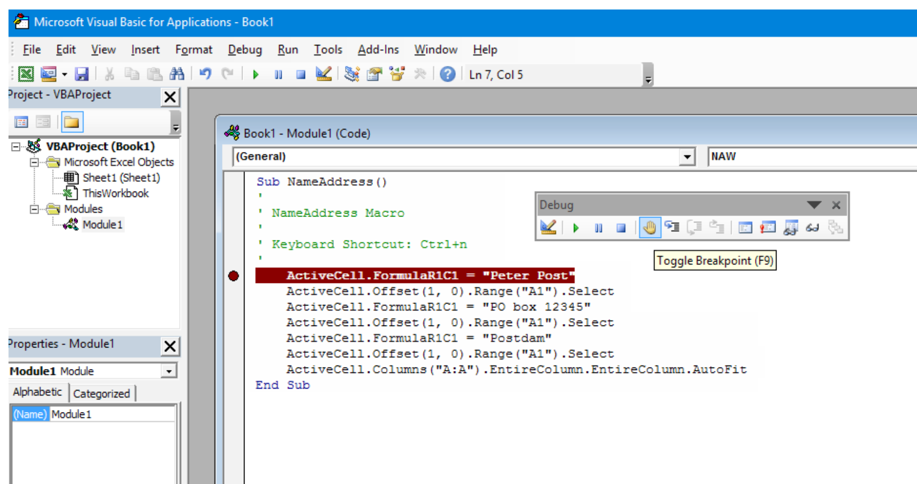


Figure 3.10 Insert Breakpoint

Download free eBooks at bookboon.com

Make sure your text cursor is on the first line of code, and click the *Toggle Breakpoint* button from the *Debug* toolbar. The line becomes highlighted red and a small circle appears in the margin.

Here we will halt the execution of the macro, and continue with the instructions step by step. You can then closely follow what is happening.

Start the macro by clicking the *Run Sub/UserForm* button from the *Debug* toolbar.

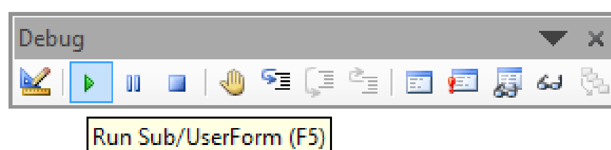


Figure 3.11 Running the macro from the code window

Notice the shortcut keys in the tooltips of the buttons that appear when hovering over them. The macro could have been started with shortkey [F5], the breakpoint could have been placed with [F9], etcetera.

The macro is immediately interrupted as the breakpoint is on the first line of the executable code in the sub(routine). The line is highlighted in yellow with a yellow arrow pointing at it shown in the margin. In this way, Excel indicates that this line is the next to be executed.

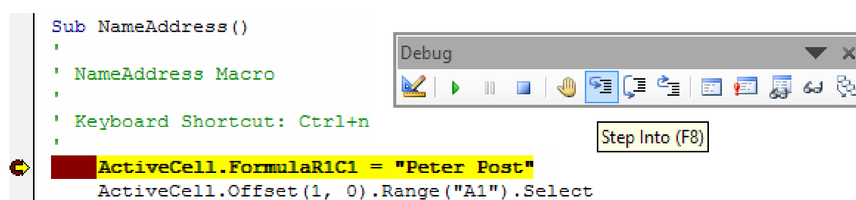


Figure 3.12 Step-by-step execution

Now proceed by executing one more step of the code. You can do this by clicking the *Step Into* button in the toolbar palette, but the shortcut [F8] is more convenient.

Now you should see that the name has appeared in the active cell in the worksheet, and that the next code line that will be executed is highlighted.

Download free eBooks at bookboon.com

Continue with the following step so that the cursor is moved down one cell.
Execute all instructions step by step and you will see the address appear and the column width being adjusted.
Ensure that the final step is also executed: `End sub` (if not the program will 'hang').

If you get 'stuck' in the program anyway and you cannot edit the code directly, you can always stop the execution of the sub with the blue square button in the *Debug* toolbar.

Change in the lines of code – not in the worksheet – the PO box number to '54321' and re-run the macro from the code window. First press [F5], immediately followed by [F5] to execute the rest.
Check the result.

So, you now have seen some lines of code, and you were even able to modify the code yourself. In macros you can capture all sorts of actions and modify them if desired or required. This too is a form of programming, albeit a modified form. A programmer would prefer to start with a blank page and enter all the code himself.

In both cases – editing macro-recordings or writing code – it is important to understand what programming is. The next chapter will deal with this extensively. Keep in mind that we will only create small programs inside of another program.

3.5 EXERCISE

Create another address macro in this workbook (named *Address*, keyboard shortcut [Ctrl]+[a]), but now without using relative references.
See what happens when you re-run this macro.
View and compare the VBA code of the two macros.

Download free eBooks at bookboon.com

```
Sub NameAddress()
'
' NameAddress Macro
'
' Keyboard Shortcut: Ctrl+n
'
    ActiveCell.FormulaR1C1 = "Peter Post"
    ActiveCell.Offset(1, 0).Range("A1").Select
    ActiveCell.FormulaR1C1 = "PO box 54321"
    ActiveCell.Offset(1, 0).Range("A1").Select
    ActiveCell.FormulaR1C1 = "Postdam"
    ActiveCell.Offset(1, 0).Range("A1").Select
    ActiveCell.Columns("A:A").EntireColumn.EntireColumn.AutoFit
End Sub
Sub Address()
'
' Address Macro
'
' Keyboard Shortcut: Ctrl+a
'
    ActiveCell.FormulaR1C1 = "Peter Post"
    Range("A2").Select
    ActiveCell.FormulaR1C1 = "PO box 12345"
    Range("A3").Select
    ActiveCell.FormulaR1C1 = "Postdam"
    Range("A4").Select
    Columns("A:A").EntireColumn.AutoFit
End Sub
```

Figure 3.13 Relative or absolute

Close your files (no need to save).

The distinction between relative references and absolute references is important because they have very different consequences when executing your programs. We will deal with this regularly in this workbook.

Download free eBooks at bookboon.com

4 PROGRAMMING

4.1 INTRODUCTION

VBA is – like its 'big brother' Visual Basic – an object-oriented and event-driven programming language. The user of a spreadsheet containing a VBA program (module) may click on buttons and options in random order and manner, insert or select specific information in boxes, etcetera. All these objects must therefore have the proper specifications and instructions independently of each other. The instructions have to be executed when a certain user action (event) occurs.

This is the reason that a VBA program is in fact made up of a whole range of small programs (subroutines) that each have their own set of instructions that can be executed independently. But modules (parts) of the program can also be linked, in one or various ways, so that the execution of instructions in one part of the program may have an impact on the operation of the result in another subroutine.

Deloitte.
Discover the truth at www.deloitte.ca/careers
© Deloitte & Touche LLP and affiliated entities.
Download free eBooks at bookboon.com
Click on the ad to read more

4.2 CREATING PROGRAMS

Before you start programming (writing instructions), it is wise to think about how and why a program is created. Consider what the program should do, what means or elements can come in handy, whether it all will function properly, etcetera.

We will now elaborate the most important aspects of creating a VBA program. Such a programming project consists of several stages.

Step 1: Problem definition

In this first phase, you should determine the purpose of your program. In case you write the program for other users, you will have to establish what these users want from the program. Say, for example, you are given the assignment to design a 'mortgage calculation program'. What does this mean? What calculations should be performed with the program and why is it not possible to achieve this with the standard calculation functions in Excel?

A possible answer to this could be: the user wants to calculate the monthly payments for a loan or mortgage but is not familiar with the functions and formulas in Excel to do that. Or maybe the result of the calculation needs to be presented to the user in a certain fashion.

After you examined and determined all wishes and needs of the users (or your own), it is a good idea to define this as a sort of starting point in a special design document, called the *program requirements*.

Step 2: Technical analysis

Once you know what has to be programmed, you will have to get familiar with the techniques and resources that are required for writing the program in what is called a *technical analysis*. Maybe you have to use a VBA language element that you have never used before. It is wise to study these codes and elements for usability in small test programs.

Step 3: Availability

Before you start programming, you need to determine how and where the program should be made available to the user. Or do you want to link it to a particular (type of) calculation? In Excel you can make programs available from within:

Download free eBooks at bookboon.com

- the parent program (so that they are always available when Excel is active),
- a specific workbook (only available when the file is in use), or
- a template (available in all workbooks based on the template that contains the modules).

During the development of a program it is good practice to link the program to a file only. This keeps Excel 'clean' from programs and lines of code that are not finished yet and it allows you to take the program with you (to other computers) to continue developing it.

Step 4: Interface

What will be the look & feel of your program experienced by the user? We call this the *interface*. Make a sketch of the window you want to show the user and think of appropriate names for the elements and controls you are going to use. The example of the mortgage calculator mentioned above could lead to a design as shown in Figure 4.1.

Calculation Monthly Costs
Calculate easily what your gross and net mortgages are with the required loan amount. You can choose from different types of mortgages and mortgage interest rates

Date of Birth
Your annual salary
Gender — Male Female
Desired mortgage amount
Desired type of mortgage — Interest Only mortgage
Death Coverage — 0 %
Real Estate Valuation — 0
Mortgage interest New mortgage — %
Continue

Figure 4.1 Example of a user interface

Step 5: Algorithms

Make a plan of all the things the program should do in case the end user uses a button, option or other element in the interface.

Begin by writing the algorithm for each subroutine in pseudo-code (a form of instructions in 'natural language', but no actual programming code). This pseudo-code can later be

inserted as remarks and descriptions in the program code. Think most carefully about the critical algorithms where the 'real work' in your program is being done.

An example of an algorithm in pseudo-code for some of the instructions in the program that you could write for the mortgage calculation form (Figure 4.1):

- The user has clicked the [Continue] button. Check if all mandatory boxes have been filled out.
- IF false THEN:
  Notify the user (preferably only about what is left blank).
- ELSE:
  Continue to the next page.

Step 6: Writing

This is where the actual programming begins, where it comes down to the dots and commas. In any case, you must ensure proper naming of all objects and variables in your program.

Also, be sure to add comments so others, or you yourself at a later point in time, can understand the code. The program is actually the coherent collection of objects, variables and instructions that perform the tasks that you have defined in step 1.

Step 7: Testing

Test the limits of the program variables, the boxes in which the user can type or choose data, etcetera. Try out all different sequences of clicking buttons or boxes in the interface, and so on. Predict what the outcome(s) of the program should be and verify that. Also allow other (potential) users to test your program! Make your programs fool-proof and bug free…

Step 8: Implementation and deployment

Now it's time to make the final program available to the users. They have to be able to determine that the program meets the requirements that were defined in step 1, known as *acceptance testing*.

Download free eBooks at bookboon.com

Don't forget to document the program and/or to create a manual. The latter can be a document (on paper), or a help function that you have built into the program.

In practice, many programmers – especially when writing small programs – immediately start with step 6 (writing). However, a good problem exploration and definition (phase 1), and thinking about the adequate algorithms (step 5) are still required: it prevents unnecessary extra work coming your way during programming.

For the next example you're allowed to conform to the real world of programmers: all you have to do is write the code.

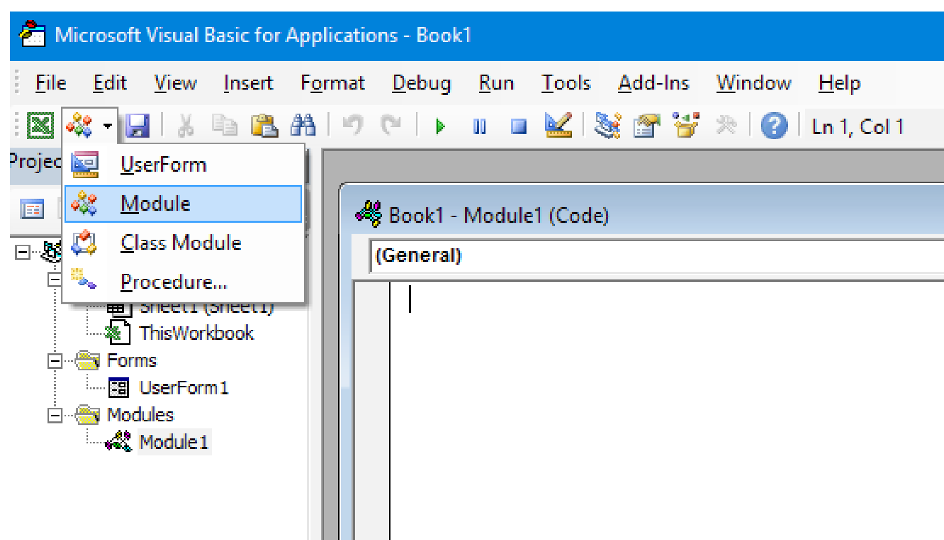


Figure 4.2 New Module

Start a new (empty) workbook.
Switch directly to the VBA Editor (use the *Developer* tab, *Visual Basic* button; however, using the shortcut key combination [Alt]+[F11] is much faster).
In the standard toolbar below the menu, click on the selection arrow of the second button and open a new module.
Type the following code in the module window:

```
Sub Welcome()
  MsgBox "Hello World!"
End Sub
```

Make sure that the text cursor is located somewhere within the subroutine and run the sub (for instance by using [F5] or the button on the toolbar). If your program works, a dialog window will appear in the worksheet.

Download free eBooks at bookboon.com

Figure 4.3 Traditional starting text for every programmer

4.3 OBJECTS

VBA projects use objects provided with program instructions. Let's look into that more closely. You can describe an object as a 'thing with a purpose'. For example, let's consider a door. Such a thing has certain characteristics, like its color or the material it is made of. The door can also do certain things, for instance "open" when someone pushes the door handle down.

In fact, the door is an object with properties and methods. Properties describe how the object looks, methods determine what the object can do. A method is activated when something happens – when an *event* occurs – that requires the object to react.

Although there are many different doors in color, type of material and size, everyone recognizes such an object as a door. Apparently there is a kind of blueprint for doors: we call that the *class* of doors. The class 'door' defines the generic properties that describe a door, like color and size, and methods such as open, close or lock. This can apply to any door.

4.3.1 PROPERTIES AND METHODS

Windows programs use many objects such as buttons and listboxes. Many of the features and actions within windows are similar, so if you use a program for the first time you will intuitively know how to make a window larger or smaller (which is a method of the window). There are also less obvious objects used in Windows applications: in Excel, for example, each cell – or even a range of cells, an entire sheet, a row number or column letter that you can click on – is an object with certain properties and methods. Think of properties such as color, format or font of the text, hidden or protected content, and of methods like delete, move or activate.

Download free eBooks at bookboon.com

An Excel workbook itself is also an object. Properties of this object are, for instance, the creator and the file name, but also the sheets it contains. And a font is an object too with properties such as name, size, color and other display options.

An example of object-oriented thinking in Excel: what exactly do you do when you make the text in a cell bold?

1. You select the cell with the text (event).
2. The selected, active cell is highlighted with a colored border (reaction of an object to an event).
3. You click on the Bold button (another event). The button looks as being 'pressed' (method).
4. The text in the cell is made bold (method of the button that changes the property of the cell).

Let's look at another example of properties, methods and events. The figure below shows three buttons (objects), each with at least one property: the caption (the text on the button).

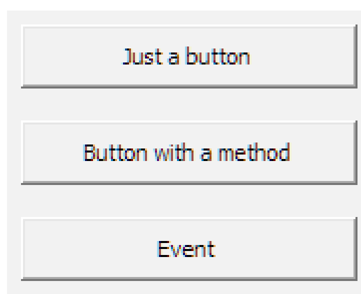


Figure 4.4 Some command buttons

The programmer who created these command buttons could have programmed them as follows:

- The first button has no method and hence no instructions have been added to this button; so it will do nothing.
- The second button responds 'normally' to clicking the button: you'll see the button being pressed.
- The third button also reacts to something, but not to the pressing of the button. It reacts, for example, when the mouse pointer comes near to the button by showing a message.

Download free eBooks at bookboon.com

The main advantages of programming with objects:

- There are many standard objects available in the programming environment that you do not have to program yourself.
- You do not always have to design and code an object from scratch. You can re-use existing or modified standard objects in other projects.
- In your programs, you will only see your own programming instructions; the code that is responsible for the properties and methods of standard objects is not visible.
- Objects help you develop code in a modular way. You basically program one step at a time: design the buttons, objects, etcetera one by one, and then build the full program by bringing them together in the project step by step.

4.3.2 OBJECT-ORIENTED

Every Office application – including Excel – is technically a collection of objects that you can make use of to create programs. These objects have properties and methods that can perform calculations (and the corresponding actions). All these properties and methods can be accessed via VBA. The objects with their properties and methods can be found in the *Object Browser* of the VBA Editor (we will discus this later).

There are a lot of objects, properties and methods. Some examples: the `Application` object (which is the program Excel itself), the `Range` and `Selection` objects (for selected cells), and the `Workbook` object. The `Application` object, for instance, has a property `Application.ActiveSheet` that contains the name of the current worksheet, and a method `Application.Quit` that closes Excel.

In VBA, you can access or often change the contents of the property of an object. The following VBA code shows how you can retrieve the name of the active worksheet, put it in a text variable and display it in a dialog window.

```
Sub SheetName()
  Dim Sname As String

  Sname = Application.ActiveSheet.Name
  MsgBox Sname

End Sub
```

Download free eBooks at bookboon.com

In these instructions the so called 'dot notation' is used: `Objectname.Propertyname.Property` (also possible for methods). To find out what properties and methods (and possibly more parameters or attributes) belong to an object, you can use the Object Browser.

While entering the code earlier, you might have noticed that the VBA Editor wants to help you in your choice of instructions. VBA will do that as well with properties and methods when typing object names: after typing a dot at the end of an instruction you will see a list of all the options that can follow the code that you already typed. This is called *hinting* or *code completion*.

Enter the new subroutine in your current module window (see Figure 4.5). Run the sub ([F5] or the button on the toolbar).

If your text cursor is outside of the module that you want to run, for example a few lines lower, Excel will show a window with the subs in a list of … macros. In that case, select the macro and click [Run] (or double-click the Macro name in the list).

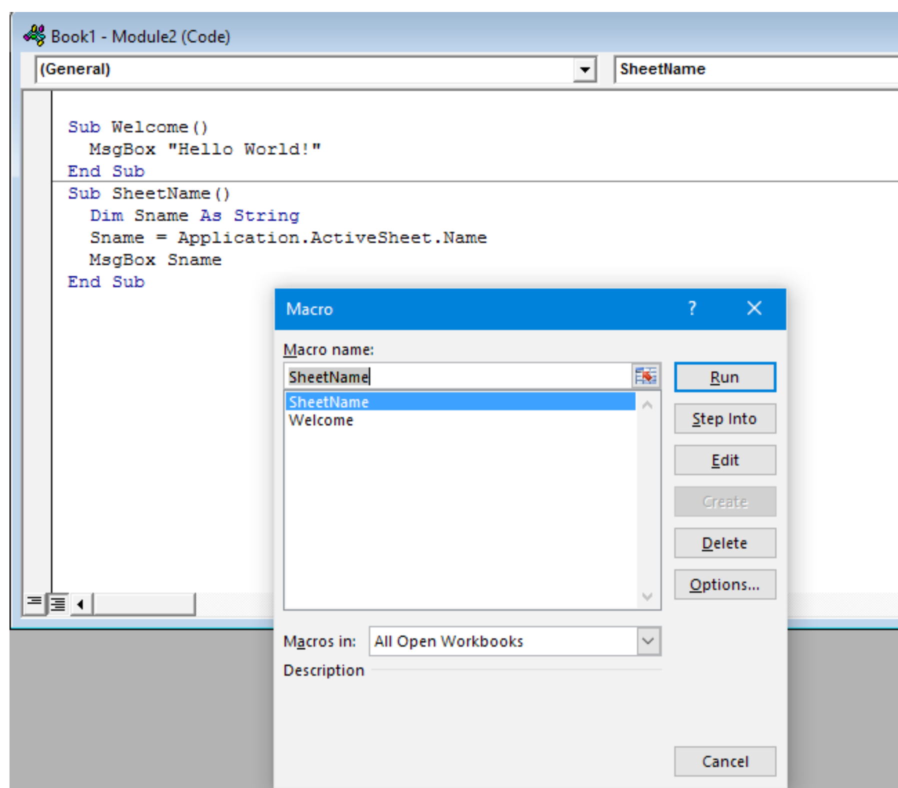


Figure 4.5 What sub to run?

If all went well, you should now see on your screen a dialog box with the name of the worksheet.

Download free eBooks at bookboon.com

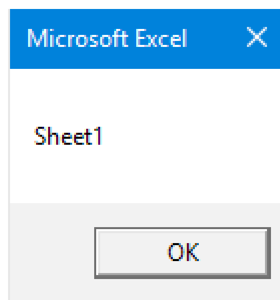


Figure 4.6 Name of the worksheet in a dialog

4.4 PROGRAM STRUCTURES

When developing your projects, it is necessary to design algorithms to make your program do what it is expected to do. Next, you convert your algorithms into pieces of program code (subroutines) and link them to the objects you are using. This is the actual writing of the program. Your choice of algorithms and objects determines the structure of your program.

The following scenarios are quite common in program structures:

- Sequence: performing operations or actions in sequence.
- Repetition: repeatedly performing (a particular set of) actions.
- Choice: the execution of actions depends on a selection, condition or choice.
- Partition: dividing the actions in parts.

4.4.1 SEQUENCE

The common way of executing statements is in sequential order. In the following program, the instructions are executed in sequence:

```
Name1 = txtLastName.Text
Name2 = txtFirstName.Text
FullName = Name2 & " " & Name1
…etcetera
```

In the first line, the last name that is entered in a textbox called `txtLastName` is stored into the variable `Name1`. In the next line, the first name (from another textbox) is put in

the variable Name2. Then a variable named FullName is composed by concatenating (&) the value of Name2, a space (" ") and the value of Name1. Variables are very important in programs as you can see. Observe that literal text (the space in this example) has to be 'escaped' between double quotes, otherwise VBA will treat it as code.

4.4.2 REPETITION

A simple way of repeating actions can be programmed in VBA by using a For-Next-loop. Say, for example, you'd have to write out 100 lines as a punishment, you could use the following code:

```
Sub Punishment()
  For LineCount = 1 To 100
    ActiveCell.FormulaR1C1 = "I shall not play games during IT-class"
    ActiveCell.Offset(1, 0).Range("A1").Select
  Next LineCount
End Sub
```

Download free eBooks at bookboon.com

Click on the ad to read more

In the For statement there is a counter variable LineCount with an initial value 1 and – after To – final value 100. After the instructions that have to be repeated, a mandatory Next statement increases the counter and the statements between For and Next are repeated. This continues until the counter has reached its final value of 100. We'll get back to these program structures later.

4.4.3 CONDITIONS

A choice in your algorithm often leads to a conditional execution of actions: IF a condition is true THEN do this, ELSE do that. You may build the next complex If-Then-Else construction in VBA:

```
If Temperature > 20 And FanOn = False Then
  FanOn = True
  ElseIf Temperature < 10 Then
    BurnHeater = True
  Else
    BurnHeater = False
Else
  If FanOn = True Then FanOn = False
End If
```

To avoid complex, nested or many repeated If constructions you can often use a Select Case statement instead. We will come to that later.

4.4.4 PARTITION

In order to split up the program tasks into parts, you use subprocedures and functions in VBA. When a button has to react to a mouse click from the user, VBA itself will generate the subroutine for that, as you might have figured out already. You can identify subs as follows:

```
Sub ProcedureName()
  … Set of instructions …
End Sub
```

You can transfer values to a subprocedure and state what is to be done with them, or what calculations are to be made, for example:

Download free eBooks at bookboon.com

```
Sub TellTime (Hour As Integer, Minute
As Integer, Second As Integer)
  … instructions that display a time with three values …
End Sub
```

A function looks similar to a subprocedure, but there is a difference: a function always returns a value to the program. An example:

```
Function CalculateVAT(Amount As Double) As Double
  CalculateVAT = Amount * 20%
End Function
```

The instruction between `Function` and `End Function` uses the name of the function again: this sets the value that the function has to return to the main program. In the first line it is specified what sort of variable the function will return as a value, in this case a `Double`. Functions are very useful when you need to perform similar calculations or actions in multiple parts of your program.

4.5 VARIABLES

A variable is technically a reserved space in the internal memory of the computer that is given a name, in which a program can store information. Variables are useful in your program to perform calculations, use as conditions in loops or represent settings. The value of a variable can be entered by the user, but can also be retrieved from a cell or calculated with a formula.

When declaring variables, you not only need to specify a name but also the type of variable, such as String (for text), Integer, Long or Double (for numbers), Boolean (True/False) or Date. If you do not know yet, you can also select the type Variant, but that requires more memory space and is considered bad practice as it can have any (unexpected) type of value.

Make it a (good) habit to explicitly declare all variables in VBA by including the `Option Explicit` statement at the beginning of your modules, even before the first subprocedure.

A variable can be declared as follows:

```
Dim <variable name> As <variable type>
```

Basically, you are free in assigning names to your variables, but it is wise to choose logical and understandable names that describe their purpose. This way other people will understand your program better and it is also useful to yourself when you want to change something in your program at a (much) later stage.

When variables are necessary only in a particular situation or part during the execution of the program, you may declare them as local variables of a subprocedure or function. If you want to use them throughout the whole program, you need to declare them at the very beginning of your program in the *General* section (before the first sub or function).

Not only variables but also constants (a fixed value) and arrays (a matrix of variables), can be useful in your program. We discuss this in more detail if it is applicable.

4.6 COMMENTS

When you write a program, make sure that it is comprehensible to other programmers. In order to do this, you can add comments to the programming code. These remarks can be very useful to yourself as well, for instance when – after a long time – you might have forgotten why you have chosen a specific construction or set of instructions in the program.

In VBA you can use complete lines to write comments, but you may also add remarks at the end of the lines of code. In the code example below you can see how we added some comments in a function (green text).

```
Function CalculateVAT(Amount As Double) As Double
  ' Calculation of the VAT amount
  ' Input: the amount over which VAT has to be calculated
  Dim VATpercentage As Double
  VATpercentage = 0.2       ' VAT 20%
  CalculateVAT = Amount * VATpercentage
End Function
```

Remarks are always preceded by a single quote ('). The comments are ignored by VBA when you run the program. The commentary serves only as documentation of the program. The

Download free eBooks at bookboon.com

second and third lines in the example above contain no code and are only comments. In the fifth line, the code is followed by a short comment relating to the code on the same line.

You can, for example, document the following topics in the comments at the beginning of a subprocedure:

- A description, in one or two lines, of the purpose of the function or subprocedure
- The variables that serve as input.
- The return value.

In some cases it can also be useful to add a short comment of one or more words at the end of a code line or just before a complex code block.

In this workbook we use empty lines to make the code easy to read. Those empty lines are used to indicate what blocks of code belong together. Empty lines are ignored by the program.

Do not be stingy with comments. It is better to have too many remarks than too few. Good documentation will save you from having to figure out what a piece of code actually does.

To design and build your programs, you have to work with the VBA Editor in your Office-application(s). This development environment is the subject in the next chapter. Later on in this book, when it's time for the real programming, all concepts and topics mentioned here will be elaborated extensively.

Download free eBooks at bookboon.com

5 THE VBA EDITOR

5.1 INTRODUCTION

VBA programs are called projects. They are built in the Visual Basic for Applications editor of Microsoft Office. The window of the VBA Editor looks outdated, because the interface is designed as in older Windows programs: you have to work with a menu with submenus, and toolbars with buttons. In this chapter we take a look at the main features and elements of this editor.

5.2 WHAT'S IN THE WINDOW?

Before we get started with creating and editing VBA projects, it is useful to study the various items in the editor window.

Create a new workbook and switch directly to the VBA Editor (fastest way: [Alt]+[F11]).

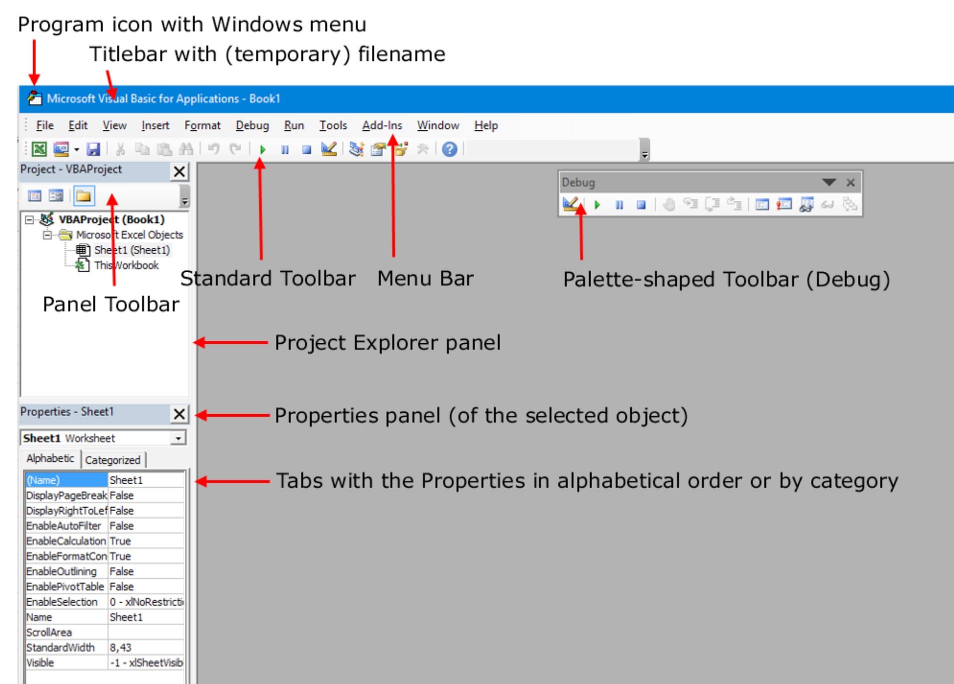


Figure 5.1 Main window of the VBA Editor

In Figure 5.1, you can see all the elements and panels of the VBA Editor shown by default at first activation. There's one extra element: the *Debug* toolbar on the right (shaped as a palette) that we activated in an earlier stage.

Note that the leftmost button in the Standard Toolbar contains the Excel logo. Clicking this button will return you to the program itself (in this case Excel). When you start the VBA Editor from a different program such as Word or PowerPoint, the logo shown will be of that program respectively.

You'll probably have to get used to working with menus and toolbar buttons, instead of the ribbon. But it is not necessarily more difficult, so you'll be able to figure it out. Buttons on toolbars are always an alternative (faster) way to choose or perform certain menu actions (often with predefined settings). By hovering over a button, a tooltip containing more information about the function or the name of the option appears. In the submenus, you can see available keyboard shortcuts for many of the menu options.

So there are basicly three ways to perform a specific action: by using the menu (always possible), with a keyboard shortcut or by clicking a button on a toolbar. Not all menu options have a shortkey and/or an alternative button. Whichever method you find most convenient, is a personal choice. Some VBA programmers do everything with hotkeys, and rarely use a button or menu option. This, of course, is convenient when your hands are already on the keyboard because you are typing code.

Let's perform a few simple actions now so that you can get used to the old-fashioned interface of the VBA Editor.

Close the *Properties* window pane with the [x] button in the upper right corner.
Close the *Project Explorer* pane as well.
Make the *Properties* panel reappear with the button on the Standard Toolbar.
Now make the *Project Explorer* reappear, using the keyboard shortcut (find out what shortkey you can use in the *View* menu or from the tooltip of the toolbar button).
Activate other window elements and toolbars from the *View* menu, and see where they pop up.

Download free eBooks at bookboon.com

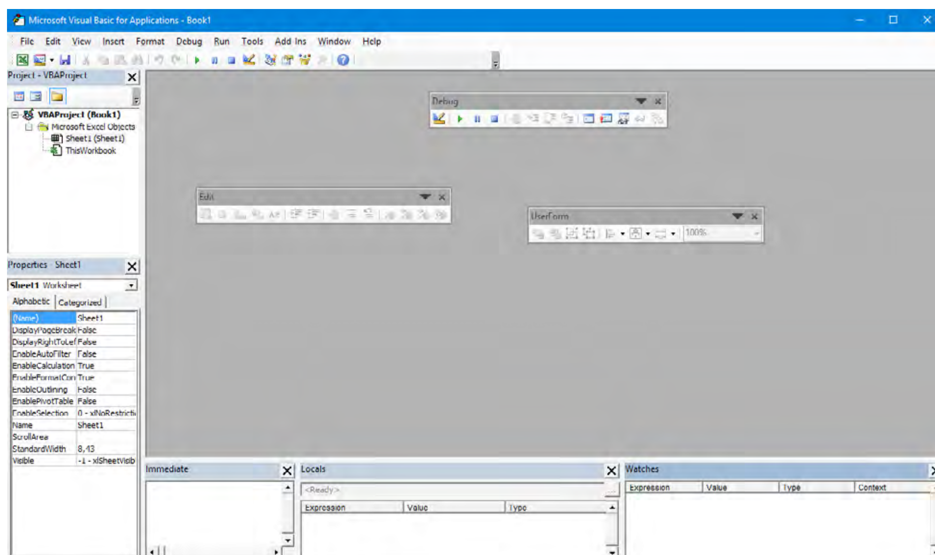


Figure 5.2 More window elements

The window panels *Immediate*, *Locals* and *Watches* are located at the bottom and toolbars appear in the window as floating palettes. So it can be crowded inside the editor window, and all at the expense of the space where you can type your code. Let us get all these elements out of the way for now, we will make them reappear when needed.

Close the *Edit* and *UserForm* toolbars, as well as the three window panels at the bottom (that leaves the situation as shown in Figure 5.1).
Click and hold the title bar of the *Debug* toolbar palette and drag it into a position next to the *Standard* toolbar. Please note that the shape of the toolbar will change when you reach that position (see Figure 5.3).

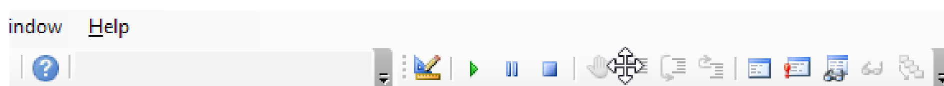


Figure 5.3 Moving a toolbar

The dots on the left side of a toolbar are called the 'handle'. With this handle you can move or place the toolbar anywhere in the area with the menus and toolbars, or drag it out of there to display and use it as a floating palette.

Download free eBooks at bookboon.com

5.3 WORKING WINDOWS

A large part of the VBA Editor window can be filled with working windows where you can do the actual work: typing code and designing your program. There is a variety of working windows:

- Module or coding window
  This is where the code that has to be executed in subroutines or functions is located. You can enter code yourself, or work with the code generated for recorded macros. A combination is also possible: you may record a macro and then add your own code to the code that was generated.
- Class Module Window
  Similar to a normal module window, but meant for the design of classes.
- Form Window
  For designing forms and dialogs. They are called VBA UserForms, and they can be composed of all kinds of controls such as command buttons, radio buttons, listboxes, etcetera.
- Object Browser
  A documentation library containing all available objects with their properties and methods.

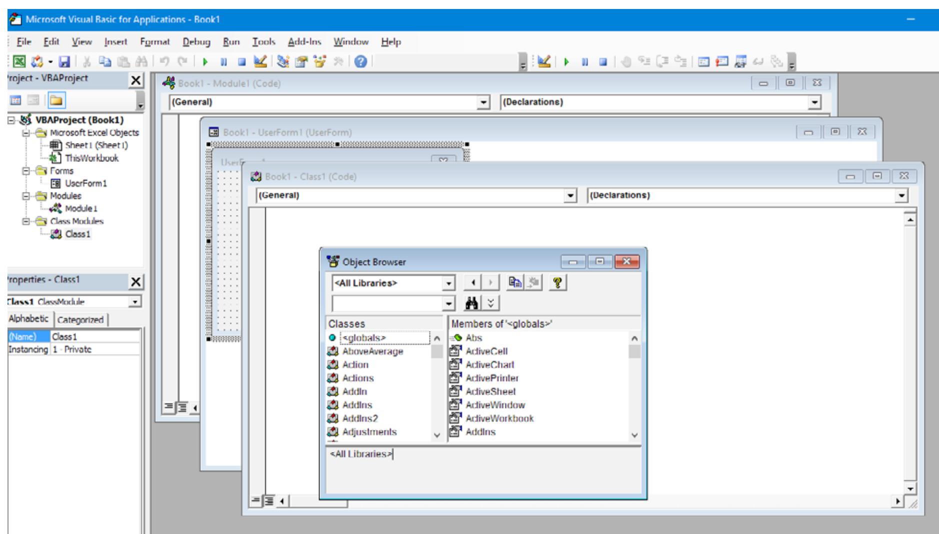


Figure 5.4 All working windows together

Figure 5.4 shows all the working windows listed above. The title bars of the windows tell you what they are called. Look in the *Project Explorer* window in the left pane to see where each element is contained within the structure of the project.

Download free eBooks at bookboon.com

If at any moment coding windows or modules are not (properly) visible in your working area, you can (re)activate a window by double-clicking a module in the *Project Explorer* pane.

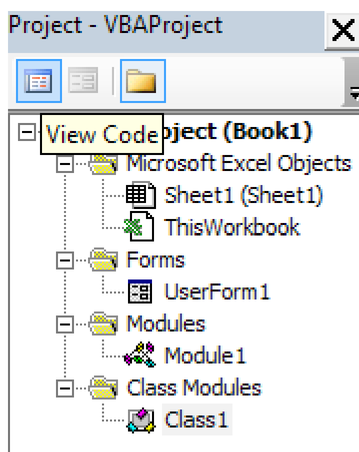


Figure 5.5 What do you want to see?

Futhermore, in the *Project* pane there is a small toolbar to toggle between different views, for example view objects or code. Make yourself comfortable working with the *Project Explorer*.

What if you could build your future and create the future?

The innovation accelerator

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".

Alcatel·Lucent

www.alcatel-lucent.com/careers

6 GENERATING CODE

6.1 INTRODUCTION

Programming is an artful craft, a real profession, and not everyone has got the talent for it. But this book is not intended to teach you how to become a programmer. We'll show you how to use tricks and built-in tools to automate certain tasks in Excel, and how you can add some extra functionality to the spreadsheet program. So that you will be able to work with the program easier and faster in practical situations.

6.2 RECORDING

Once more we begin with recording a macro, not only because it saves a lot of typing but also because you don't have to look up what statements you need and what their correct usage and syntax is. Please perform the following instructions carefully.

Start with a new workbook (close any other).
Activate the VBA Editor ([Alt]+[F11]).
In Windows, place the two windows side by side on the screen.
In the window of the workbook go to the *Developer* tab, check the option *Use Relative References*, and start a new macro recording.
Name the macro *WorkWeek* (stored in this workbook), assign the keyboard shortcut [Ctrl]+[w] to it, and set as *Description*: Place the working days in a range of cells.
Click [OK], but do not take any further actions yet!
Activate the VBA Editor window and open, using the *Project Explorer* on the left, *Module1* from the *Modules* folder (double-click). The working window will appear, maximize it within the Editor window. In this window you'll see what code is generated during the recording.

Now switch back to the worksheet in Excel. Enter in cell A1 (which is the active cell) the textlabel *Monday*: type it in the cell or the formula bar, but finish by clicking the check-icon in the formula bar (so you don't move your cell pointer).
Look at the code that appears in the Module window.

Use the Fill Handle (Copy Tool, the square in the lower right corner of the cell border) to add the next four days of the week in the range A1:A5. The code that appears in the Module window is probably not what you had thought of yourself.

Download free eBooks at bookboon.com

After the AutoFill with the series of days, the range of cells is still selected. Set the following formatting features for the range: bold and right aligned.

If you make a mistake while recording your macro, you can simply undo the action and its code is immediately deleted in the Module window. Try it with any one of the formatting settings (and redo it of course).

Adjust the Column Width to autofit the longest working day textlabel.
Then select the cell right next to textlabel *Monday* (because that's probably the position you want to continue working in your model).
Stop the recording.

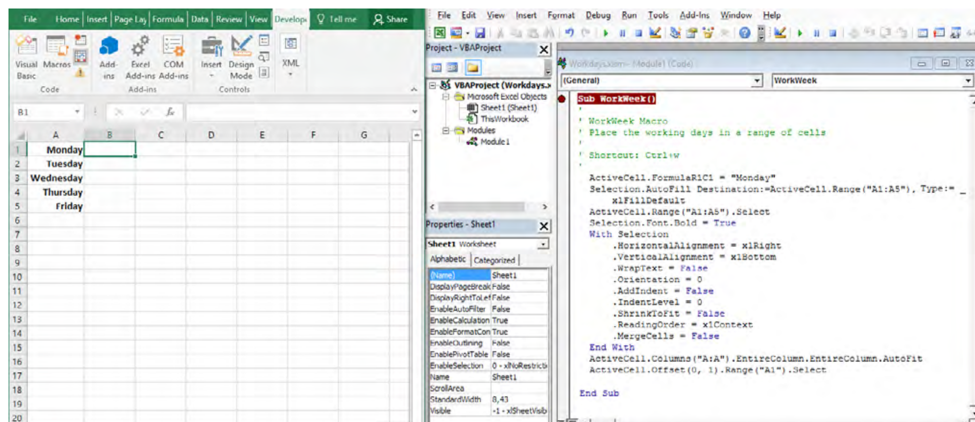


Figure 6.1 Code generated during macro recording

By now the program should have generated the code shown to the right in Figure 6.1. Quite a lot of lines – that you would not have thought of yourself – are generated by Excel just because they are the settings for your cell range.

Note that at the end of the line that controls the AutoFill a space + an underscore _ forces the (long) instruction to continue on the next line. You can always place this 'breaking character' when your instruction does not fit on one line. If not placed in the correct position, an error message will automatically appear.

Select a different cell in the worksheet.
In the VBA Editor, place a breakpoint on the first line of the subroutine (like you earlier have done in Chapter 3, see Figure 6.1).
Run the macro now step by step (begin with [F5] and step forward with [F8]).

Download free eBooks at bookboon.com

With each line of instructions that is executed, you see the result in the worksheet. With the exception, of course, of quite a number of instructions in the program construction `With Selection … End With`. Besides an action corresponding to setting the property `.HorizontalAlignment = xlRight`, nothing happens to the other settings. In fact those instructions could be removed.

6.3 EDITING CODE

The code in your Module window can be modified, so deleting the redundant instructions mentioned previously is no problem.

Before removal, you can mark (potentially) unnecessary lines of code as comments – by typing an apostrophe as the first character of the line – so they are ignored when running the macro. This technique is very useful when you are not quite sure if the program will run properly after deleting the lines.

Remove the redundant lines of code from inside the `Selection … With` construction.
Run the macro again (preferably step by step so you can see what happens). Is the result the same? If not, you probably have deleted too much (use undo to retrieve the removed lines).

The `With` construction is of course very useful if you want to set several properties for the selected cell(s) at the same time. However, when all you want is right alignment, the one-line alternative is more practical:

```
Selection.HorizontalAlignment = xlRight
```

which is in fact the same as:

```
With Selection
  .HorizontalAlignment = xlRight
End With
```

In both situations, don't forget to separate the property from its object by a dot. If it's not there, the code will result in an error.

Download free eBooks at bookboon.com

Outside of the `With` construction there is yet one more setting for the selection: the instruction to make the textlabels bold. You may choose to also set this property within the `With` construction (look carefully at the dots):

```
With Selection
  .Font.Bold = True
  .HorizontalAlignment = xlRight
End With
```

Edit your code as above and test the macro.
Save your workbook (*Macro-Enabled*) and name it **Workdays**(.xlsm).

You have now saved a clear piece of program code in your workbook that you can run as a macro. But it's also possible to add some more interaction: let's, for example, ask the user in a dialog what day to begin with.

Add the two lines of code shown in red below. They must be typed before the instruction `ActiveCell…` that puts the text in the first cell. Replace the text `"Monday"` in that line by the variable `txtDay` (whose content is supplied by the user through the `InputBox`).

```
Sub WorkWeek()
'
' WorkWeek Macro
' Place the working days in a range of cells
'
' Shortcut: Ctrl+w
'
  Dim txtDay As String

  txtDay = InputBox("What is the first working day?", "Settings")

  ActiveCell.FormulaR1C1 = txtDay
  Selection.AutoFill Destination:=ActiveCell.Range("A1:A5"), _
      Type:=xlFillDefault
  ActiveCell.Range("A1:A5").Select
  With Selection
    .Font.Bold = True
    .HorizontalAlignment = xlRight
  End With
  ActiveCell.Columns("A:A").EntireColumn.EntireColumn.AutoFit
  ActiveCell.Offset(0, 1).Range("A1").Select
End Sub
```

Download free eBooks at bookboon.com

The first inserted line is the declaration of a variable that you need to store the text that the user types in the dialog (the inputbox in the second line). The information stored in the variable will next be entered in the subsequent line as the starting textlabel for further proceedings. Later we will discuss the use of variables more extensively.

Run the subroutine ([F5]). The input dialog appears with the title and the question: enter a day of your choice.
Click (in the inputbox) the [OK] button. If there is still a breakpoint in the code, the program will now halt here. Press [F5] again to continue the procedure. After that you can remove the breakpoint ([F9] with your text cursor on that line or by clicking on the circle in the margin).
The result is a list of working days, starting with the day the user (in this case you) entered in the dialog in the worksheet.

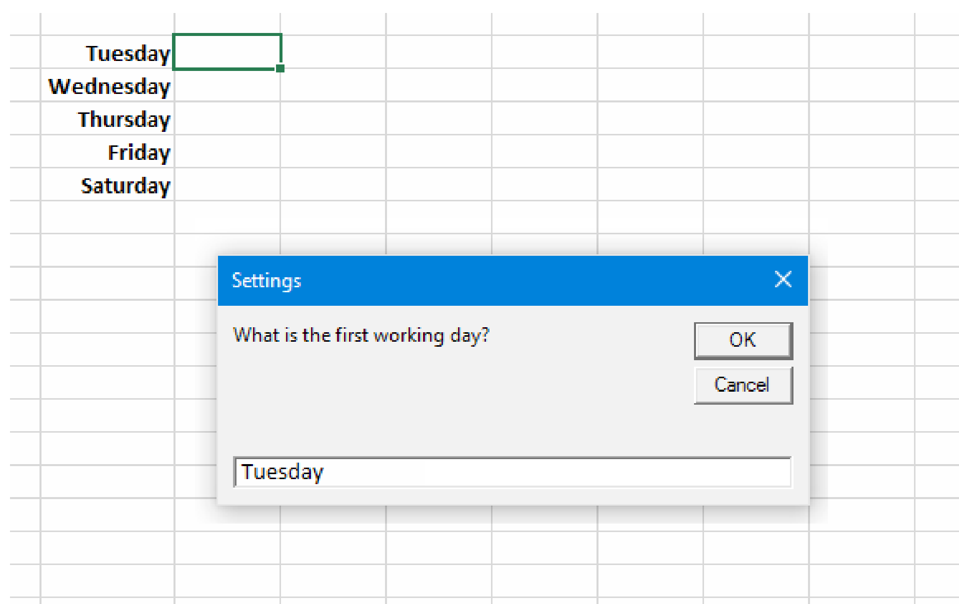


Figure 6.2 Input by the user

This of course is working nicely, but be careful: you never know what the user will type in the inputbox. Actually any textual input is possible, and whatever text is typed here will be the starting value for the AutoFill. And knowing the possibilities of this function you will probably be aware of the consequences that the input can lead to.

Run the program a few more times with as input: fr (short for Friday), Product1, Quarter1 and January. Also try entering a text like 'my day off' or a location such as 'London'. Please note what happens with the Column Width each time you enter new input.

It's possible to tackle such cases in certain ways before continuing further execution of the instructions in the program. But you might find it more convenient to limit or prevent these things beforehand. You can do this by using a form, instead of a simple input dialog, that enables you to take (full) control of the interaction.

6.4 FORMS

Besides (several) modules, a VBA project can also contain forms. A form can have many properties and (control) elements, and program code can be provided to both the form and the elements in it. This allows you to create user friendly dialog windows, but it takes a lot of programming. Let's see what we can achieve with a form.

Add a *UserForm* to the project (using the the selection button on the toolbar).

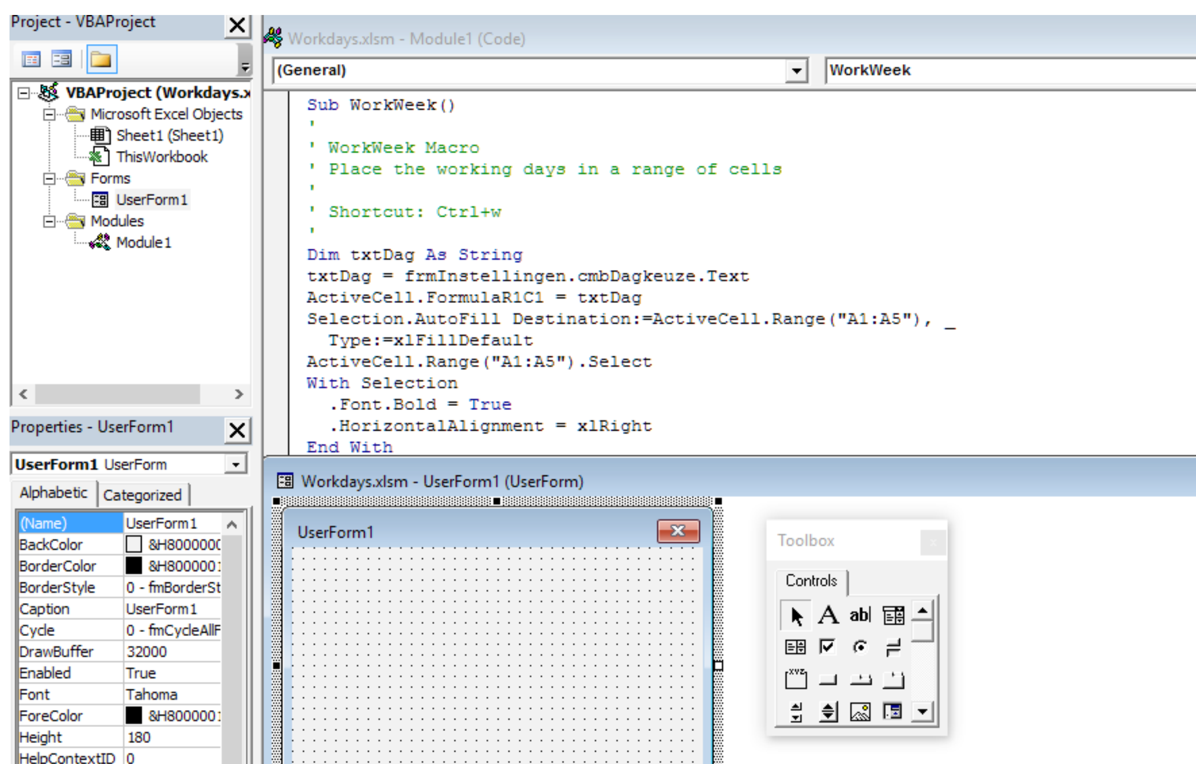


Figure 6.3 New UserForm

In the window pane *Properties* on the left, change the name of the form to *frmSettings* (the name will change in the *Project Explorer* accordingly) and the *Caption* to *Settings*. This caption appears in the title bar of the form.

Download free eBooks at bookboon.com

In VBA, any form is an object of the UserForm *class*. It can hold other objects and controls. For example you can place a command button on the form as we demonstrate in the next exercise.

Make sure that the Toolbox with form controls is nearby (in Figure 6.3 we have placed it next to the form). This toolkit appears when the form is active (in case this does not happen, click on the button in the toolbar to make it appear).
Draw a command button in the lower right corner of the form: click the appropriate control in the Toolbox and drag a rectangular shape on the form grid.
In the *Properties* pane, name this button *cmdExecute* with the *Caption* text *Create WorkWeek*. Change the width of the button if the text does not fit in.

As you may have noticed by now, in this book all names of objects begin with a short code indicating the type of object. That's why we named the command button *cmdExecute*, and the form *frmSettings*. With these prefixes you can easily recognize the type of objects, even if you use many elements. This naming convention is known as the 'Hungarian notation'.

Double-click on the command button. A new code window opens for the selected object.
Insert the instruction `Call WorkWeek` between the two lines of code. Please don't use brackets, in fact, if you would type them, Excel removes them.

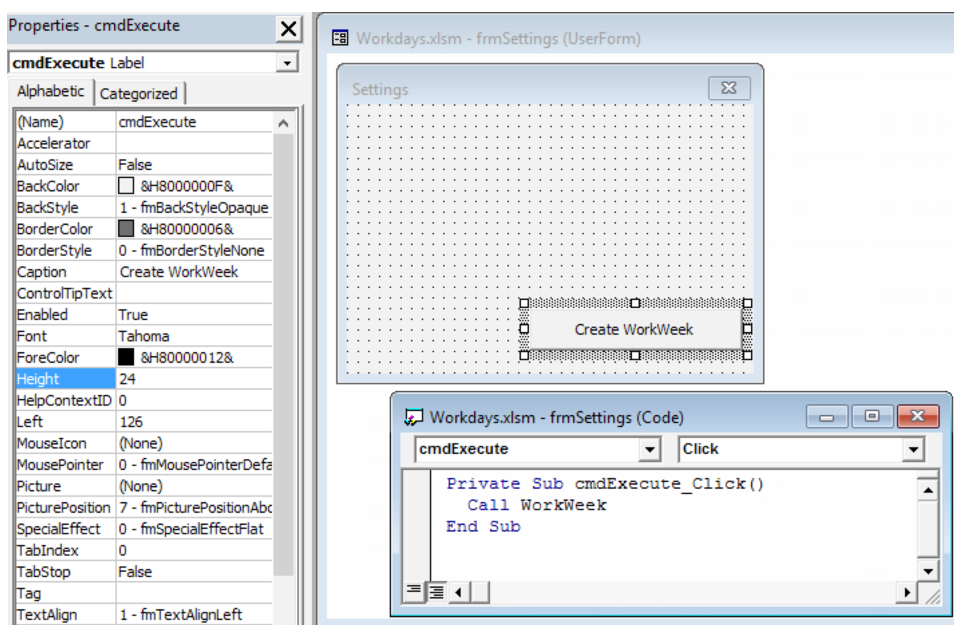


Figure 6.4 Command button with code

Download free eBooks at bookboon.com

Objects like the button – usually called controls – have all kinds of properties and methods that allow handling of events involving the objects. For example you can set what should happen when – during execution of the program – the user clicks on the command button.

In the top of the code window there are two listboxes. The left-most box contains a list of objects and the other a list of events. These listboxes tell you that you're working in the subroutine `Private Sub cmdExecute_Click()`.

This subprocedure is a `Private Sub`, which is a different kind of subroutine than the `Sub WorkWeek()`. Notice that the word 'Private' is missing there and that's why it is called a *public sub*. A private sub is only usable within the module in which it is defined, for instance, the code for the button is only for use in the form. A public sub is also available for use from within other modules. In that case, we can use the `Call` statement followed by the name of the public sub (see Figure 6.4). Now let's see if it works.

Start the program from the private sub, in other words: activate the form. This will show the form in the worksheet.
Click the [Create WorkWeek] button. Then of course you should type your starting day and continue.
Insert another series of working days using the button on the form.

As long as the form is visible, a new working week is put in the column right next to the previous series. That's because in the sub WorkWeek you move the active cell pointer to the cell right of that of the starting day. In between, you can not navigate to another cell in the worksheet. For that you must close the form.

Close the form *Settings* (using the standard Close button).
Go back to the design window of the form (for instance by double-clicking in the *Project Explorer*).
At the bottom of the form, add a button named *cmdClose* to [Close] the form (don't forget to set the caption). The instruction you need here is: `Unload Me`.
Test the form with the two buttons.

As you can see, there are several controls in the *Toolbox* that you can use on your form. Most of them you'll probably already know from your experience with dialogs and windows in other programs: checks, textboxes, radio buttons, labels, option and selection lists, and the like. Let's look at a few of the options so you can see how they can be incorporated into your program.

Download free eBooks at bookboon.com

To determine the starting day, you could use a textbox with a corresponding label. But this comes with the same 'problem' as when using the inputbox: the user is completely free in what he enters. In the next exercise we will therefore limit his choice to only the days of the week.

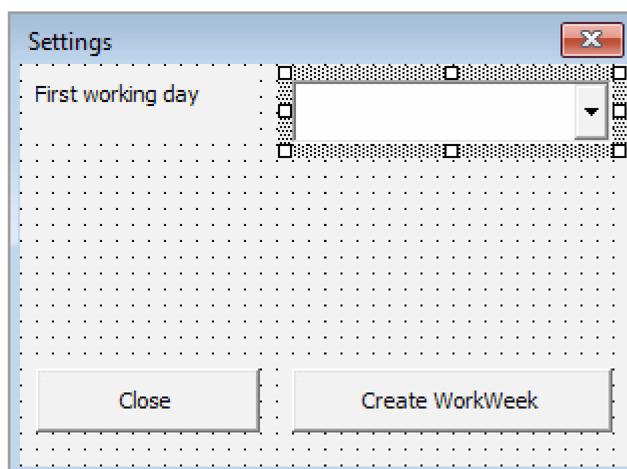


Figure 6.5 Textlabel and combobox

Place a label in the top of the form and edit the *Caption* to match the figure shown above. Insert a combobox right next to it. In the *Properties* panel, change the names of the objects to *lblFirstDay* and *cboFirstDay*.

With the *AddItem* method we can provide options to the listbox control in the form. You'll want to do this, of course, every time the form opens, so we will link the action to that event. The form is also an object to which all sorts of events can apply, like the opening event (`Initialize`) that occurs when the form is shown on the screen. That is typically a good moment to set properties of objects.

Create the `Private Sub UserForm_Initialize()` in the code window of the form (if necessary, double-click an empty spot on the form to open the code window). Instead of typing the code you can use the listboxes at the top, but then don't forget to remove any redundant code.

Download free eBooks at bookboon.com

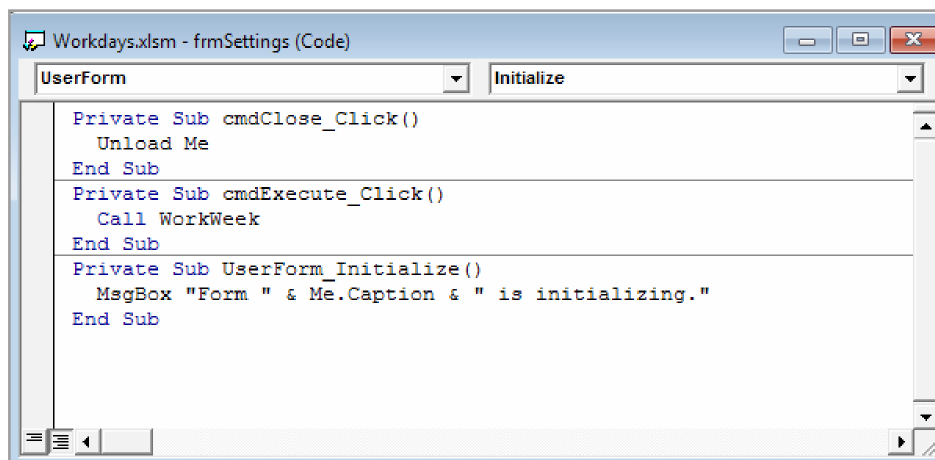


Figure 6.6 Code when opening the form

In the code window, add the code for a dialog as shown in Figure 6.6. Test the form.

So, that's settled: you have told the user now that first the program is setting up some things for the form. As you can see, despite of the position of the sub (defined as third), this procedure is executed first, even before the form itself appears.

Most programmers 'feel' better when the code to be executed first comes first in the code window. You can move the subs around if you like.

Notice the reference in the code to the form itself: we can use `Me` instead of the full code to refer to the form, like in `frmSettings.Caption`. Because this sub is run when the form is already loaded in the background, it is not necessary to address the form by its name in full.

Using the following code, you can fill the listbox with the days of the week.

```
' Listbox cboFirstDay values
frmSettings.cboFirstDay.AddItem "Monday"
frmSettings.cboFirstDay.AddItem "Tuesday"
frmSettings.cboFirstDay.AddItem "Wednesday"
frmSettings.cboFirstDay.AddItem "Thursday"
frmSettings.cboFirstDay.AddItem "Friday"
frmSettings.cboFirstDay.AddItem "Saturday"
frmSettings.cboFirstDay.AddItem "Sunday"
```

You might refer here to the form name with `Me` as well and you can copy and paste much of the code. You also know a different technique to formulate this nice and easy: using a `With` construction. Before you continue reading the text, try to do the next exercise all by yourself.

Add the necessary code to the sub `UserForm_Initialize()` to fill the combobox with items using a `With` construction.
Test the form.

Your code should be something like this:

The Wake
the only emission we want to leave behind
Low-speed Engines Medium-speed Engines Turbochargers Propellers Propulsion Packages PrimeServ
The design of eco-friendly marine power and propulsion solutions is crucial for MAN Diesel & Turbo.
Power competencies are offered with the world's largest engine programme – having outputs spanning from 450 to 87,220 kW per engine. Get up front!
Find out more at www.mandieselturbo.com
Engineering the Future – since 1758.
MAN Diesel & Turbo
MAN
Click on the ad to read more

```
With Me.cboFirstDay
  .AddItem "Monday"
  .AddItem "Tuesday"
  .AddItem "Wednesday"
  .AddItem "Thursday"
  .AddItem "Friday"
  .AddItem "Saturday"
  .AddItem "Sunday"
End With
```

We will now arrange that the choice from the listbox is used in the subprocedure `WorkWeek`, instead of the text entered via the inputbox. We use a variable `txtDay` for that and assign the text of the selected item to it using the following line of code:

```
txtDay = frmSettings.cboFirstDay.Text
```

Edit the code in the sub `WorkWeek()` to include the above line – you can remove the inputbox code – and test the form again. Be careful: because you're outside the form now, you can no longer refer to the form name with `Me`!

As long as the form is open, this will function properly because the value of the listbox is available. But closing the form after placing the series of days can be useful if you want to move to a different location to add another series. To close the form together with the creation of the series, you need to add the code `Unload me` **after** the workweek routine is completed.

Add the code to close the form to the button `cmdExecute`.

Everything will work nicely now, but still a user can type anything he likes in a combobox. This may come in handy, for example when typing the first letter(s) to quickly select the day of your choice. But it also means that arbitrary input is possible, just try it. Changing the *Style* property will prevent that free input is entered.

Set the *Style* property to `2 – fmStyleDropDownList`. To change the property of the control, it must be selected in the form.

Download free eBooks at bookboon.com

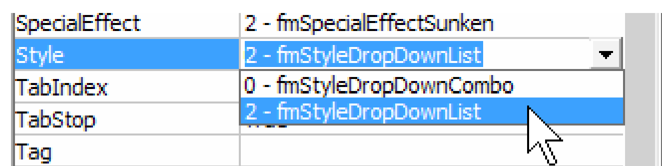


Figure 6.7 Changing Style of drop-down list

With this *Style* setting only items from the list can be chosen, although you may still type in the box.

As you can see, there are many configuration options available regarding the form and its controls. For instance, think about the following issues:

- Which element should get first focus after opening the form?
- What should be the tab order (index and stops) when navigating in the form with the [Tab]-key?
- What is the function or behaviour of the [Enter]-key?
- …etcetera.

Please study the list of properties and options of the various control elements thoroughly.

Save your workbook (including the code), you'll need it later. If you now continue with the next chapter, you do not have to close the file.

Download free eBooks at bookboon.com

7 OVERVIEW

7.1 INTRODUCTION

Generating code is one thing, keeping a clear overview is a different matter. But, this is a very important one because all the subs that you implement in your program may soon lead to a complete tangle of code fragments (also called 'spaghetti code'). Fortunately, the VBA Editor has several tools to help you keep a clear overview. For instance, the *Immediate* window (in which hinting is active), the *Object browser* and other control panels. The occasional step-by-step execution of code, as we did before in chapter 3, also helps you to keep your overview of the project. The usual troubleshooting method is also a good way to keep an overview of everything.

Maintaining a good overview starts in your programming: make sure to give the objects (and variables) clear and comprehensible names so that you always know what they are and what they are doing within your program. Before we continue to the actual programming in the next chapter, let us first explain a bit more about these overview topics.

7.2 HUNGARIAN NOTATION

In object-oriented programming, it is necessary to refer to the various objects frequently, so it is very important that you name everything properly. It is a good idea to use a naming convention such as the *Hungarian notation*. This entails that each object is given a name that starts with (usually three lower-case) characters to indicate what type of object it is, or to what object it belongs. The type indicator is followed by an actual objectname (preferably starting with a capital) that describes the function of the object or its intended use.

You may use more than one capital in an objectname, see for instance `cboFirstDay` in the figure below. This is called 'camel-casing'.

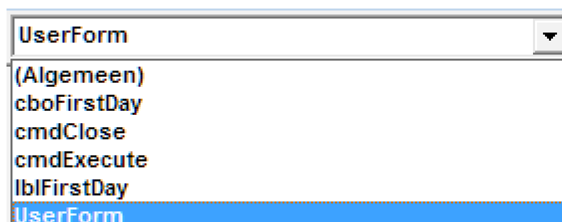


Figure 7.1 Objects in your form

Download free eBooks at bookboon.com

If necessary, open the file **Workdays**, and go to the VBA Editor.
Examine the top-left listbox in the code window of the form *frmSettings*. Here you can see the elements that you have placed in the form (in chapter 6), populated with names in accordance with the Hungarian naming convention.

Using this notation you can easily recognize, retrieve and link the objects. Another advantage of this method is that in the code window the items in the listbox with objects will be presented alphabetically, sorted by their type indicator. Usually English prefixes (character codes) are used, the table below shows the names and type indicators of frequently used objects.

Object	Naming
Label	`lblName, labName`
Textbox	`txtName`
Combobox	`cboName, cmbName`
Listbox	`lstName`
Checkbox	`chkName`
Option Button	`optName`
Command Button	`cmdName`
Form	`frmName`

Properties and methods can also be provided with a (usually two-character) type indicator, you have seen this at the end of the previous chapter with the *Style* property of the combobox: `fmStyleDropDownList`. A property like this was not designed and named by you, but when you create your own classes you could choose to name them in this manner.

Some programmers add prefixes to all of their variables (in one or more lower-case characters) sometimes just to indicate that it is a variable – for example, `varCounter` or `vCount` – and in other cases, in order to make the data type of the variable identifiable – as in `curRate` (variable of type Currency) or `intCounter` (type Integer). In this workbook we do not use indicators in the names of variables.

Download free eBooks at bookboon.com

What's Hungarian about the notation?
Originally the idea of naming objects with a prefix that conveys extra information was invented by the Hungarian Charles Simonyi – who, in Hungary, is called Simonyi Károly. A bit similar to the reverse order of first and last names in Hungarian, an objectname like 'Settings form' can be reversed (and shortened) to 'frmSettings'. The Hungarian notation is also referred to as the Leszynski Naming Convention (LNC).

The use of type indicators is not mandatory or required, but the additional information may well provide much clarity and insight. The code will be better readable and moreover, coding this way is perceived as very professional. Another advantage is that similar objects have the same prefixes and therefore appear together in listboxes or insertion lists when using hinting.

7.3 IMMEDIATE WINDOW

Hinting, which is when the editor shows you what objects or options you could use in the current situation, is available in almost all windows in which you write code. You can also experience it in the *Immediate* window, a special window pane that you can use for debugging without having to mess around in (properly functioning) code.

Open the *Immediate* window (use the button on the *Debug* toolbar or the shortcut [Ctrl]+[g]).
Type in: `frmsettings.` (including the dot). After typing the period a list will appear.

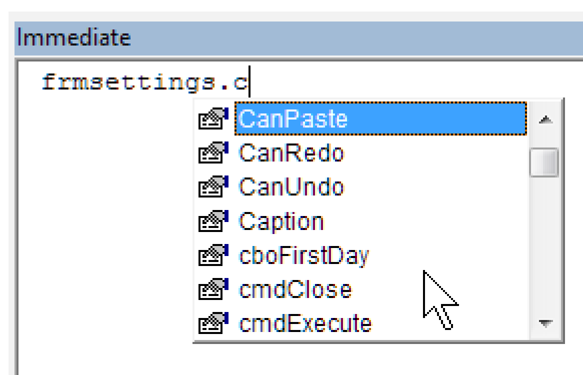


Figure 7.2 Hinting in window Immediate

This list includes all objects, properties and methods associated with *frmSettings*. Because of the Hungarian names you used, all command buttons, radio buttons, etcetera, are grouped together in the list. So, for instance, continuing typing with the c you can relatively easily locate the objects, features and options starting with c that are available.

Download free eBooks at bookboon.com

7.4 OBJECT BROWSER

Another way to find objects, items and associated options is through the *Object Browser*, also called the *Object viewer*. You can open this overview via menu *View*, *Object Browser*, by clicking the button on the standard toolbar or using shortcut [F2].

Everything in the *Object Browser* is grouped into libraries. When you select one on the top left, Excel shows you all classes this library consists of and what members belong to these classes. The class *UserForm* is a general object with a series of properties such as Caption or Name. When you create a new form in VBA, it will be based on this general class *UserForm*.

The form you have created in the example in the previous chapter is hence an object based on the general class *UserForm*. However, it differs from other forms belonging to this class in its specific property values and the contained controls. Morover, beside the class-defined members, each form can have additional members on its own. These are the objects on the form, such as buttons, text and listboxes, and the like, but also the implemented subroutines and functions to deal with its events.

Open the *Object Browser*.
Choose the *VBA Project* item on the top left (where you can look into <*All libraries*>) and see what classes exist in this library.
Select the class *frmSettings* (which is your form), and view all members of this class. You will encounter your ‘own’ elements such as the listbox, the buttons and the methods too.

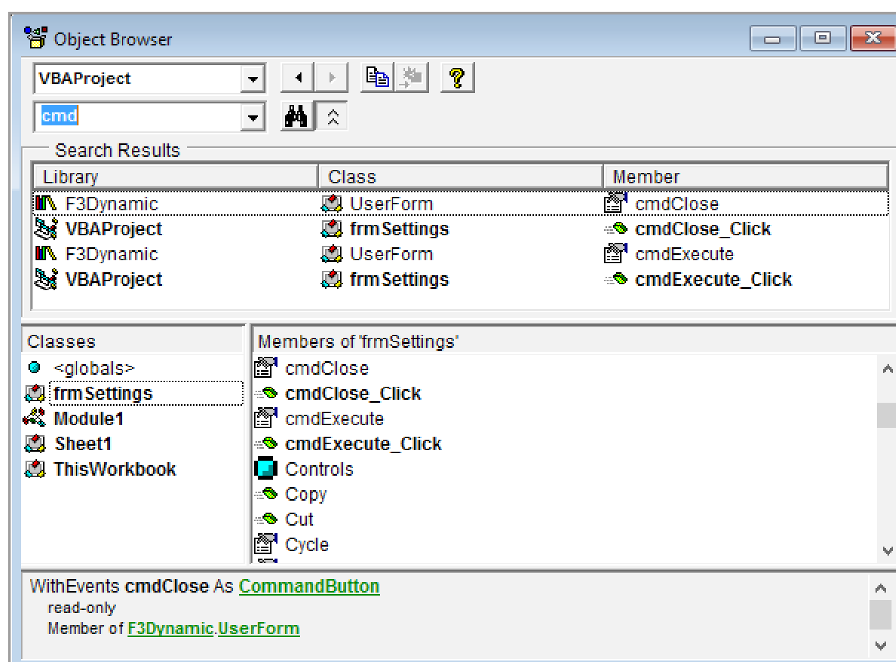


Figure 7.3 Object Browser

Now type 'cmd' in the box to the left of the search button (the one with the binoculars) and click on that button to display the search results, containing all command button objects.
Click on the class <*globals*> and, after briefly studying its members, select the class *Module1*. The sub *WorkWeek* appears to be a member of both classes.

Because the *WorkWeek* subroutine is a member of the class <*globals*> you can call it from any desired location in the program code.

Double-click member *WorkWeek* in the class <*globals*> to open the corresponding code window.
Change the subroutine now into a `Private Sub`. As soon as you move the text cursor to a different code line, the procedure will disappear from the class <*globals*> in the *Object Browser*. If you have both windows on screen, you'll be able to see it happen, otherwise you'll have to check it afterwards.
Undo the change(s) in the *WorkWeek* code – so remove the addition `Private` – and close the workbook.

Once you start designing and programming, make good use of the tools described here. Also check out the help information provided by Microsoft, of course. We will discuss the use of these tools further when necessary.

7.5 EXERCISE

All VBA commands are in English – even if your Excel program interface uses another language – and therefore the English Excel terminology is used, e.g. Workbook, Worksheet, Cell, Column, Chart, Formula, etcetera.

Go to the Excel VBA Editor and open the *Object Browser*.
Search for a couple of typical Excel objects such as Worksheet, Chart or Range (use the button with the binoculars to search) and find out what methods and properties they have.
Look up a few more objects or elements that you normally use in Excel.
Close the *Object Browser*.

Download free eBooks at bookboon.com

8 PROGRAMMING TECHNIQUES

8.1 INTRODUCTION

In this chapter, we will look at more advanced techniques of programming. There are quite a few solutions to issues that you may encounter during programming while dealing with users. Usually there is more than one solution to a problem, so which one should you choose? Sometimes it's a matter of considering pros and cons, in other cases you are forced to apply one particular method in order to reach a certain result. We discuss in this section the use of (own) functions, decision structures and loops, the declaration of variables and constants, and handling of "errors".

8.2 FUNCTIONS

Most programs contain actions that have to be performed repeatedly. Think of displaying a message to the end user, or performing a calculation on certain values. In the first case, you can use a subprocedure that can be called from any location within the program.

```
Sub Notification()
  MsgBox "You have to fill in something..!"
End Sub

' The subprocedure above is invoked
Sub Invoke()
  Call Notification
End Sub
```

You may leave out the keyword `Call` but the complete instruction makes the code easier to read and more accessible to outsiders.

For the calculation you will need a (custom) function that returns the result to the calling procedure. This result is often called the *return value*.

Download free eBooks at bookboon.com

```
Function Vat(Amount as Double)
  Vat = Amount * 0.2
End Function

' The function above is invoked
Sub Calculate()
  Dim VATamount as Double
  VATamount = Vat(txtInput.Text)
End Sub
```

In this example, the function `Vat(…)` is invoked from the subprocedure `Calculate()`. The value the user has entered in the textbox `txtInput` – for example in a form – is passed on to the special variable `Amount`, as a parameter or attribute of the function `Vat(…)`. After the calculation in the function the result is assigned to the variable `Vat` that has (not coincidentally) the same name as the function. This ensures that upon returning to the sub `Calculate()` after the `End Function` instruction, the return value is transferred to the `VATamount` variable.

UNLEASHING
CHANGE
MANAGEMENT
OCTOBER 18 & 19, 2018
DE RODE HOED
AMSTERDAM
Global
Executive
Events
Click on the ad to read more

When you do not indicate what type of variable the return value should have, it will automatically be assigned the type Variant. Excel can also use this type for calculations. More on variables later in this chapter.

Functions, like subprocedures, can be Private or Public. Private means that they are only accessible from within the current module, Public functions can be called from other modules. By default, functions and subs in VBA are Public. Making functions and subprocedures private ('encapsulation') prevents that the code can be wrongfully applied to properties or methods in other modules.

Start a new workbook and go to the VBA Editor.
Create a new form with the name `frmVAT` and title (caption) `Calculate VAT`.
Insert the following elements in the form:

- a textbox `txtInput` to enter a number
- a label `lblInput` left of the entry box, with a caption as shown in Figure 8.1
- a label `lblResult` to store the result of the calculation in; empty the caption of this label
- a label `lblOutput` next to the result box (see caption in Figure 8.1)
- a command button `cmdCalculate` to perform the calculation
- a command button `cmdClose` to close the form

Download free eBooks at bookboon.com

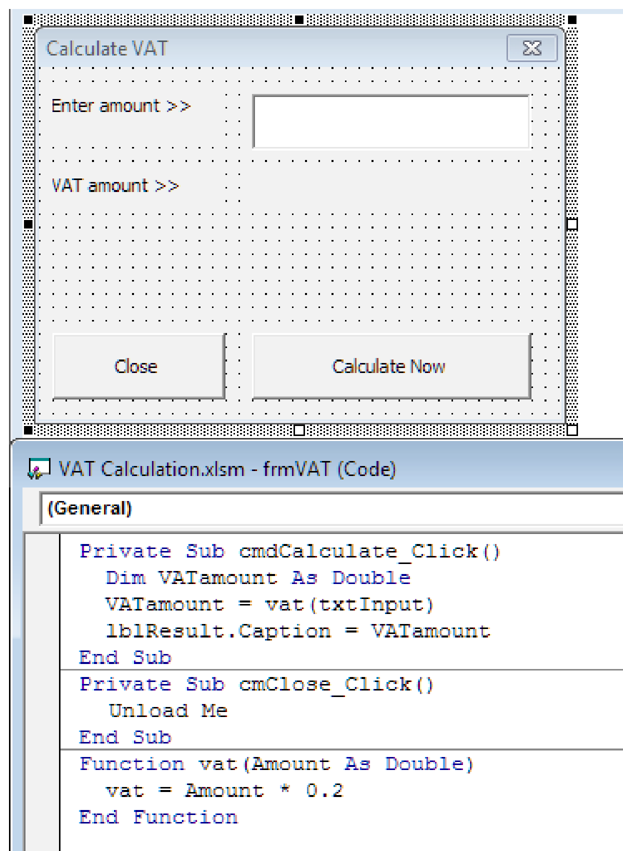


Figure 8.1 Design of a VAT-form.

Add the code fragments to the objects as shown in the bottom window in Figure 8.1. Test the form.

In case you accidentally get stuck somewhere: stop the execution of the code by pressing [End] or the Reset button (blue cube) in the *Debug* toolbar. Do not be tempted (yet) to hit the [Debug] Button. Error trapping will be discussed later.

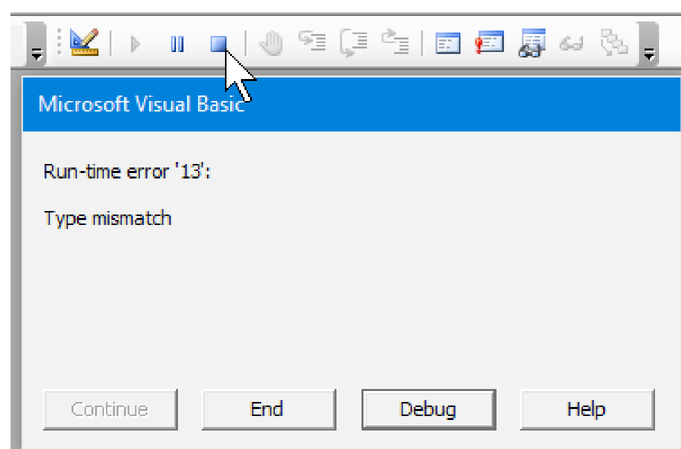


Figure 8.2 Ending the procedure after an error

The code in the form consists of two subs and one function, the subroutines for the command buttons react to the `Click` event and run the corresponding code. VAT calculation is done in the sub `cmdCalculate` using the `Function VAT(…)`.

If you want to format the result in the form as an amount with a currency sign (here we use the euro sign), you could add the `Format` function in the following way:
`lblResult.Caption = Format(VATamount, "€ #0.00")`

8.3 DECISION STRUCTURES

You can control how the program should act in certain situations by using decision structures in the code, such as the constructions `If-Then-Else` or `Select Case`. An example: when the user of your form hits the [Calculate Now] button without having entered anything in the inputbox, there is nothing to calculate. You could notify the user about that with the following code.

```
If txtInput.Text = "" Then
' when no text was entered
  MsgBox "You have to enter an amount..!"  ' notify the user
Else
' else, run the next instructions
  Dim VATamount As Double
  VATamount = vat(txtInput)
  lblResult.Caption = Format(VATamount, "€ #0.00")
End If
```

Modify the code as shown above (the comments are of course optional).
Check what happens when you hit the [Calculate Now] button without entering an amount. Also test the procedure with input to make sure everything works nicely.

Because the instructions for calculation and presentation of the result can only be executed when there is input in the textbox – the `Else`-part of the construction – an empty textbox will yield no "serious" error message besides your own message.

The `Else`-part of the `If-Then-Else` construction is optional: you can leave it out in situations where it is not required. For example in the code below the instruction to notify the user when (and what) text is entered is quite useless.

Download free eBooks at bookboon.com

```
If txtInput.Text = "" Then
  MsgBox "You have to enter an amount..!"
Else
  MsgBox "Your input was accepted."
End If
```

The redundant message has to be closed by the user every time, this only leads to frustration and a mouse arm (repetitive strain injury).

8.3.1 NESTING

An `If-Then-Else`-construction can also be *nested*: so it can be used within another `If`-statement. This nesting can be very complex, and it uses parentheses like in mathematics: you have to close each `If`-instruction with `End If` at exactly the right position in the code. The best way to write down these constructions is by using indentation of the text, as shown in the following lines of code.

```
Private Sub cmdCalculate_Click()
  If txtInput.Text = "" Then
    MsgBox "You have to enter an amount..!"
  Else
    If Val(txtInput.Text) = 0 Then
      MsgBox "The input must be a number!"
    Else
      Dim VATamount As Double
      VATamount = vat(txtInput)
      lblResult.Caption = Format(VATamount, "€ #0.00")
    End If
  End If
End Sub
```

The indentation shows what `Else` and `End If` belong to what `If`-instruction: we have highlighted the coherent parts with the same color in the code above.

Insert the extra check in the code in the form as shown above.

The second `If` -construction checks whether the numerical value of the input text is zero. This way you can try to intercept undesirable entry of text strings. Further on we will discuss a different way to check for incorrect input.

Download free eBooks at bookboon.com

8.3.2 SELECT CASE

The construction with `Select Case` is less complicated in structure than a complex tangle of nested `If`-statements, and therefore easier to read. But in turn it offers less options to construct complex conditional structures. Below you can compare two functions that basically do the same thing: the left one uses nested `If`-statements (this could be written down shorter because the third check is not necessary), the right one is a construction with `Select Case` as an alternative.

<table>
<tr>
<td>

```
Function BoxColor(Color
As Integer)
  If Color = 1 Then
    BoxColor = vbGreen
  Else
    If Color = 2 Then
      BoxColor = vbYellow
  Else
      If Color >= 3 Then
        BoxColor = vbRed
      End If
    End If
  End If
End Function
```

</td>
<td>

```
Function BoxColor(Color
As Integer)
  Select Case Color
    Case 1
      BoxColor = vbGreen
    Case 2
      BoxColor = vbYellow
    Case Else
      BoxColor = vbRed
  End Select
End Function
```

</td>
</tr>
</table>

Can you deduce from the code what these functions actually do?

The function of this code is to change the color of a box depending on a certain numeric value. For example: when in the Vat-form the inputbox remains empty, you can not only notify the user about it, but also color the box green where the input should have been entered. The first time this omission occursyou color the box green, the next time yellow, and all subsequent times red (we will implement this later).

8.3.3 COUNTER

To keep track of the number of times the user has entered nothing or an incorrect value in the `txtInput` box – as outlined above – you will need a counter. This is a variable you can increase each time the undesired input occurs. We have discussed variables before, especially the need to declare them and the scope of their use. What we need here is a variable that is active and can be increased as long as the form is open. Therefore, we declare this variable at the beginning of the form module.

Download free eBooks at bookboon.com

Create a new line in the code window, before `Private Sub cmdCalculate_Click()`; see in the option boxes at the top of the window that this is in the 'section' (General) – (Declarations). Type, on the new line, the instruction `Option Explicit` (this enforces you to always declare your variables).
On the next line enter the instruction `Dim nErrors As Integer`. If done correctly, this instruction remains above a horizontal line in the declaration zone (see Figure 8.3).
Determine where in the code the counter should be increased so that it increases when an error is made by the user (no or false input). Try to work it out yourself before looking at our solution with the necessary lines of code in Figure 8.3.

Put a breakpoint on the first code line of the sub `cmdCalculate`.
Open (when necessary) the *Immediate* window pane at the bottom of the screen ([Ctrl]+[g]).
Run the module and immediately click on the [Calculate Now] button (while the textbox still is empty).
Step through the instructions and check in the *Immediate* window the value of the counter variable by using the command `?nErrors` [Enter]. This will print its current value.
When the sub is completed, click the [Calculate Now] button again. Recheck the value of `nErrors`. You do not have to re-type this command each time, simply return the text pointer to the line with the command using the arrow keys and hit [Enter] to execute the command again. Previous results will move down.

Download free eBooks at bookboon.com

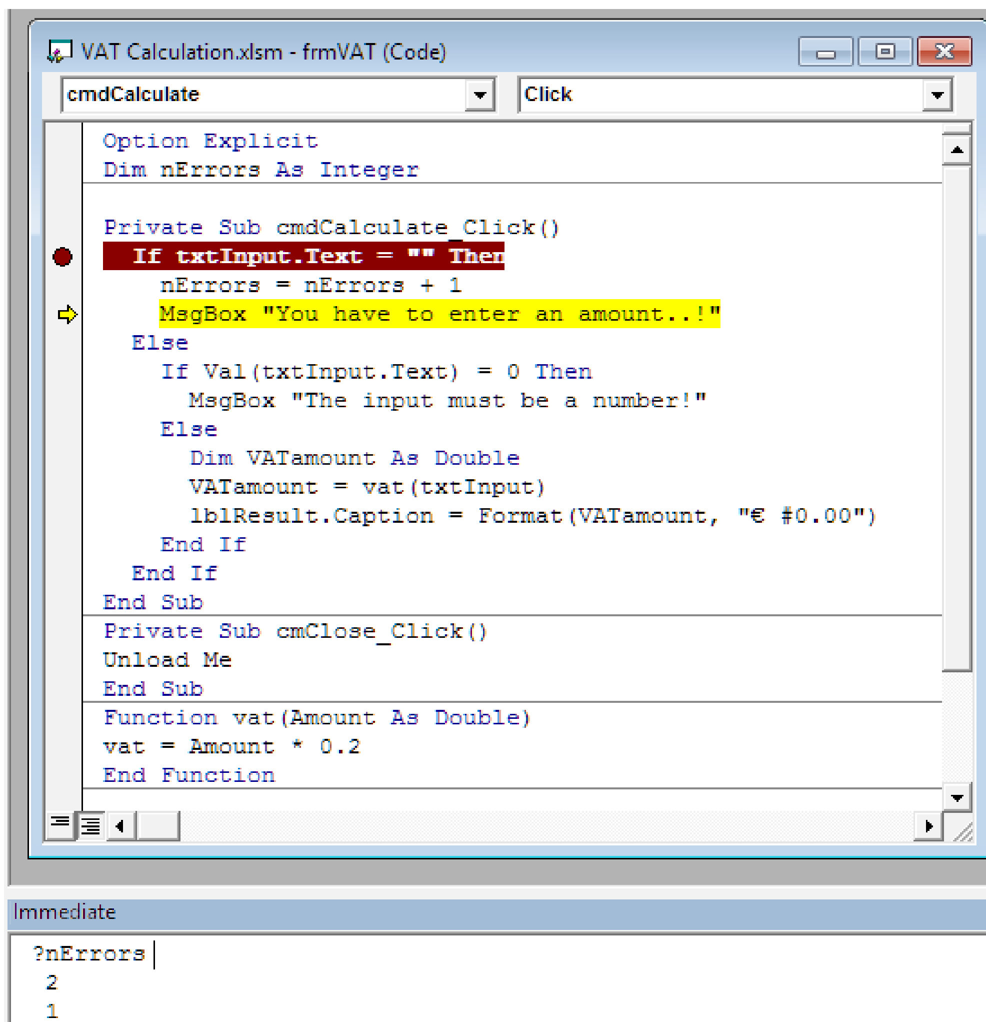


Figure 8.3 Monitoring a variable

If all goes well, the value of the counter will increase as long as the form is active. We can use the increasing value to change the background color of the textbox.

Add the function `BoxColor` at the bottom of the code window by using the code shown in paragraph 8.3.2.

Insert the following code line at the (two) required positions to set the color with the function:

```
txtInput.BackColor = BoxColor(nErrors)
```

Make sure that after the error message has been acknowledged, the background color changes back to white using the following code: `txtInput.BackColor = vbWhite`

Test the form (don't forget to calculate a VAT-value once in awhile or at the end, to see if it still functions properly).

Download free eBooks at bookboon.com

Can you still keep up? The complete code for the form you should have created by now, is shown below. In case your program is not entirely flawless, please compare your code with the instructions in our lines.

```
Option Explicit
Dim nErrors As Integer

Private Sub cmdCalculate_Click()
  If txtInput.Text = "" Then
    nErrors = nErrors + 1
    txtInput.BackColor = BoxColor(nErrors)
    MsgBox "You have to enter an amount..!"
    txtInput.BackColor = vbWhite
  Else
    If Val(txtInput.Text) = 0 Then
      nErrors = nErrors + 1
      txtInput.BackColor = BoxColor(nErrors)
      MsgBox "Your input has to be a number!"
      txtInput.BackColor = vbWhite
    Else
      Dim VATamount As Double
      VATamount = vat(txtInput)
      lblResult.Caption = Format(VATamount, "€ #0.00")
    End If
  End If
End Sub

Private Sub cmdClose_Click()
  Unload Me
End Sub

Function Vat(Amount As Double)
  vat = Amount *0.21
End Function

Function BoxColor(Color As Integer)
  Select Case Color
    Case 1
      BoxColor = vbGreen
    Case 2
      BoxColor = vbYellow
    Case Else
      BoxColor = vbRed
  End Select
End Function
```

Download free eBooks at bookboon.com

8.4 LOOPS

As we mentioned before, you often encounter situations in which certain actions have to be repeated several times. For this you can use loops that repeatedly run through a set of instructions. Below you will find some common examples of loop constructions.

8.4.1 FOR-NEXT

A `For-Next` construction always requires a counter that is incremented. You can declare these counters locally, and typically a simple `i` is used for this. You may also give the counter variable names like `varCounter`, `intNumber` or `nErrors` but this might make the lines of code unnecessary long.

In the next example we will use `i` to show in a dialog what the value of `i` is at the moment of displaying that dialog. Each of the 5 times the program runs through the loop, the value is shown and with the instruction `Next` the variable `i` is increased. When `i` equals 5, the loop ends.

```
Sub LoopForNext()
  Dim i as Integer
  For i = 1 to 5
    MsgBox "The value of the counter is " & i
  Next i
End Sub
```

Note that this is not the usual way to monitor the value or contents of a variable or object. It's more convenient to do that with the `debug.print` instruction in the *Immediate* window.

The `For-Next` loop construction is useful when you know exactly how many times a certain action has to be repeated. The number of repetitions can be a constant (as shown above) but it can also be the result of a function or calculation, e.g. the length of a string you want to examine character by character. This length can vary since you do not know beforehand how long the string will be that a user enters. You'll see another example of this later.

Download free eBooks at bookboon.com

Increasing the counter within the `For-Next` construction does not have to be in steps of 1, you can enter the step size yourself. The following examples are possible uses:

`For i = 10 to 100 step 10`	**sets the interval between steps to 10: 10, 20, 30 etcetera**
`For i = 5 to 1 step -1`	**to decrease the value of `i` from 5 to 1: 5, 4, 3, 2, 1**

You can also design a construction that allows you to exit the loop before all repetitions are done by using `Exit For`. The program then continues with the instructions in the lines of code under `Next`.

8.4.2 DO-WHILE/UNTIL

A `Do-While` statement works in a similar way as the `For-Next` loop, but there is no counter needed. This means that the loop is ended when a certain condition is met, such as when a specific value is reached or something else happens. In some cases this can come in handy, especially when you do not know exactly how many times the actions in the code should be performed.

```
Sub LoopDoWhile1()
  Dim Number as Integer
  …
  Do While Number < 5
    Number = Number + 1
    MsgBox Number
  Loop
  …
End Sub
```

In this example the program first checks whether the value of the variable `Number` is less than 5 and, if so, its value is increased by 1 and displayed in a message box. This will continue until the variable equals 5, then `Number` will no longer be increased and the loop is terminated. The program then continues with the next instruction after `Loop` (the second series of dots).

A major difference with the `For-next` loop is that in this case the initial value of `Number` might be the result of calculations in the instructions preceding the `Do-While` loop (the first series of dots). That could mean that the value of the variable does not start with 0 (its value at declaration) or 1, or even that `Number` it is not less than 5 (anymore). The

latter means that the program skips all instructions in the `Do-While` loop. Moreover, the variables used in the condition may be modified from within the loop, as we did by `Number = Number + 1`. Keep in mind, however, that in this way you can also introduce infinite loops that may never terminate and 'freeze' Excel! Make sure the condition is always met at some point (or you use `Exit Do` to exit the loop).

A variation of this loop is the construction shown below in which the condition is checked afterwards.

```
Sub LoopDoWhile2()
  Dim Number as Integer
  …
  Do
    Number = Number + 1
    MsgBox Number
  Loop While Number < 5
  …
End Sub
```

Download free eBooks at bookboon.com
Click on the ad to read more

Here too it is possible that in a previous instruction a certain value was assigned to the variable `Number`, but the instructions before `Loop While` will be executed at least once.

When using these loops you can exit the loop by using the instruction `Exit Do`. The program will continue with the first instruction after the `Do`-construction.

Instead of using `Do While` **you may also use** `Do Until` (with check before or after). **This however usually means that you have to reverse the condition:**
`Do While x < 5` **(execute as long as the value of x is less than 5)**
is equal to
`Do Until x >= 5` **(executes until the value of x is 5 or higher)**

8.4.3 WHILE-WEND

Another variant of the `Do-While/Until`-construction is the `While-Wend`-loop. This, somewhat deprecated construction, is avoided because it does not offer the possibility to exit the loop.

```
Sub LoopWhileWend()
  Dim Number as Integer
  While Number < 5
    Number = Number + 1
    MsgBox Number
  Wend
End Sub
```

8.4.4 USING LOOPS

You can try out the code examples of the previous paragraphs in a new module of a new worksheet/workbook (you have to remove the dots or replace them with valid instructions). When you run the subs step by step, you can see exactly what is happening. While executing these loops, things must be applied or can be changed, for instance, counters or variables increase or change, conditions become true or false, etcetera. There is also the possibility that certain calculations or conditions cause the loop to be exited.

What loop construction you choose depends on how many times something has to happen and whether a specific condition has to be checked at the beginning or at the end of the

Download free eBooks at bookboon.com

construction. It is wise to give it a good thought, but it can be a matter of trying what loop yields the desired result.

In the example below we add our custom validation function to the input: did the user type a (positive) number in the inputbox? We will examine the string in the textbox `txtInput` using a loop: if it turns out that any of the characters in the string is not a number or a dot, we can conclude that the string as a whole is not a number.

Add the following function under the existing code of your form.

```
Function IsValidNumber(strInput As String)
  Dim i As Integer
  For i = 1 To Len(strInput)
    Select Case Mid(strInput, i, 1)
      Case ".", "1", "2", "3", "4", "5", "6", "7", "8", "9", "0"
      IsValidNumber = True
    Case Else
      IsValidNumber = False
      Exit For
    End Select
  Next i
End Function
```

This function examines each character throughout the whole length of the string (computed by `Len(strInput)`) that the user has entered in the textbox `txtInput` and checks if it is part of the set of characters in the first `Case`-statement. If not, the function value is False and the loop will be exited, because there is no need to continue the validation. When returning to the subroutine from where the function was called, the number of errors has to be increased.

Change in the `Sub cmdCalculate` the line`If Val(txtInput.Text) = 0 Then` into`If Not IsValidNumber(txtInput) Then`
This calls the validation function and returns whether the entry is a number or not. Test the form.

8.5 KEY CAPTURING

A different way to check whether the text in the textbox is an actual number, is by using a subprocedure that monitors the actual keystrokes. Each time a key is pressed the program

Download free eBooks at bookboon.com

will check if the according character is allowed: any number or a dot. All other characters that are entered by the user are ignored and will not be added to the inputstring. This 'guarantees' that the string can only be a number.

Double-click the `txtInput` box and add a `Private sub` for the event `KeyPress` (select it from the top-right option box); Excel will generate the code items required to handle keystrokes. Delete the unnecessary sub of the `Change`-event.

```
Private Sub txtInput_Change()

End Sub

Private Sub txtInput_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)

End Sub
```

Figuur 8.4 Sub for key check

Add the following code to the subroutine.

```
Private Sub txtInput_KeyPress(ByVal
KeyAscii As MSForms.ReturnInteger)
  Select Case KeyAscii
    Case Asc("0") To Asc("9")
      ' valid input, do nothing
    Case Asc(".")
      If InStr(1, txtInput.Text, ".") > 0 Or txtInput.SelStart = 0 Then
        KeyAscii = 0
      End If
  Case Else
    KeyAscii = 0
  End Select
End Sub
```

This subprocedure will accept only characters that are allowed in the input string: if the key pressed is a number or a dot – the latter character can occur in the string only once and cannot be in the first position – it will be added to the input string. In all other situations the key will not be considered further by setting `KeyAscii = 0`.

We found this smart and useful procedure on the internet in a VBA programmers forum. Many ideas have already been invented, so when you run into a problem in your program,

chances are that someone has come up with a solution for it. And often with code that you did not know of or thought of (yet).

Save and close the workbook (name it **VAT Calculation.xlsm**). You will need it again in the next chapter.

8.6 VARIABLES AND CONSTANTS

You have seen them quite often already: variables that are required for your program, for instance to perform calculations, to run through loops or to realize settings. Let us look at the main aspects of variables.

Whether or not you have to declare your variables depends, as we mentioned before, on the instruction `Option Explicit`. If this statement is in the (General) section of a module, all variables you use have to be declared, without the statement it is not necessary. But without declaring the variables the program code can be very confusing, especially for outsiders. And what is the value of the variable when you accidentally use the same name for a variable in a different module, sub or function?

8.6.1 LOCAL OR GLOBAL

Variables can be local or global and this influences within what scope the variables are available to other modules.

- Local variables are only accessible within modules, functions or subprocedures where they are declared. You can do this at the top of a module in the (General) section before the subprocedures and after the `Option Explicit` statement (if used, see above), or right below the first line in a sub or a function using the instruction `Dim` followed by the name and the type of the variable.
- Global variables are declared at the top of a general module (not in the form), after `Option Explicit` (if used), with the statement `Public Dim`. Variables declared this way are accessible from all modules and procedures.

When possible please use local variables, because they cannot influence properties or methods in other than their own environment.

Download free eBooks at bookboon.com

8.6.2 TYPES OF VARIABLES

There are many different types of variables, just look them up in Help or on the internet. We will discus here the most used types: Boolean, Integer, Long, Double, String and Variant. When you run into other types please research the possibilities of that specific type.

- BooleanThis is a type of variable with only two possible values: the variable is True or False. Compare this to a power switch, a checkbox, or a single bit. You can use the Boolean variable to keep track if a certain action has occurred or not. When you want to display a message in the screen only once, for instance the first time something occurs, you can keep track of this with a Boolean variable. After the message is displayed the first time, the value of the variable is changed in the programming code. A decision tree can check if the window has been displayed (True) or not (False) and based on this result decides whether the window will be displayed or not.

```
Dim IsDisplayed as Boolean
…
  If Not IsDisplayed then
    MsgBox "Hello"
    IsDisplayed = True
  End If
```

  At declaration a Boolean variable is set to False, so the check in the example above results in displaying the message box. The check could also have been formulated as: `If IsDisplayed = False Then` or `If Not IsDisplayed = True Then`. In the last example you can leave out `= True` (like in the code above).
- Integer and Long
  Integer is a type of variable in which values in the range between -32.768 and 32.767 can be saved. To save larger numbers you have to use the Long integer. This type is also called Long and has a range between -2.147.483.648 up to and including 2.147.483.647. Integer variables are saved as whole numbers in 2 bytes. Long variables are saved as whole numbers in 4 bytes. Integer variables use less memory and are therefore to be preferred if its range is sufficient.
- Double (Precision)
  This type of variable can hold numbers with decimals that require a lot of precision. The range is from -1,79769313486232 e^{308} to -4,94065645841247 e^{-324} for negative values, and from 4,94065645841247 e^{-324} to 1,79769313486232 e^{308} for positive values. They are stored in 8 bytes.

- StringA String variable consist of a series of characters that are interpreted as actual symbols (so not the numeric value of the symbol). You can not perform calculations with numbers that are saved in a String variable (unless you convert them using for instance `Val`). A String type variable can contain a maximum of circa 2 billion characters.
- VariantA universal but rather vague type of variable is Variant. When you declare a variable and do not specify what type of variable it is, it will be defaulted the type Variant. Variant variables can contain numeric values, dates or character strings. Avoid using them if possible, because it is not in any case clear what they can do, contain or cause in you program.

Please note that certain operations between different types of variables are not possible. The Long variable for instance can not be combined in operations with a String variable, this could lead to a 'data type mismatch'. You can prevent this by declaring both variables as the same data type or by converting the variables with the help of (advanced) functions to the same type of variable.

8.6.3 CONSTANTS

Constants have, opposite to the variables, always the same value. They are usually declared in the (General) section, but declaration is also possible within a sub (in which case it would only be valid in that sub). Once you've declared a constant after the `Option Explicit` in a general module, the value cannot be altered in the program code. Constants are useful for instance for path links used in different locations in the program, or to set fixed values such as a VAT percentage. As situations change, and you'd want to change the path link or the VAT percentage, you only have to change the value assigned to the constant in one place: the general section. Using constants to represent values that are used in more than one place is considered good programming practice.

```
Option Explicit
Public Const conPad = "C:\test"
Sub constant()
  ' Open the file test.xlsx that can be found in the C:\test folder
  Workbooks.Open (conPad & "\test.xlsx")
End Sub
```

Download free eBooks at bookboon.com

8.7 ERROR TRAPPING

In case of an error, the VBA program halts (goes into *Break Mode*) and displays an error message. This is very useful to the programmer, but the end user cannot continue and will also be able to see the program code. It is better to add lines of code or procedures to the program to prevent the program from crashing upon such errors. We call this *error trapping*. Two examples of how this could be done.

```
Sub ProcedureName()

  On Error Resume Next

  … code in the sub

End Sub
```

The line `On Error Resume Next` at the beginning of the subroutine ensures that when an error occurs, the execution of the program continues with the next code line. The error itself will be ignored, the code only prevents the program to halt or fall into *Break Mode* and should thus be used with care.

```
Sub IsFaulty()
  On Error Goto fault

  … code in the sub

  Exit Sub

fault:
  MsgBox Err.description
  Exit Sub
End Sub
```

The second example will jump to the *label* `fault:` in the code when an error occurs. The procedure displays an error message and comes to an end. `MsgBox Err.description` shows the default VBA error message, instead you can also have your own error message displayed.

When you are not sure what will happen in the next code line after an error occurs, it is better to choose an error trap such as the one shown in the second example.

Download free eBooks at bookboon.com

8.8 EXERCISE

In the example below we will create a form to show the differences between the types of variables. We'll also build in an error trap in the coding of the form.

Create in a new workbook a new form called *frmVariables* and make it look like the form in Figure 8.5.

Figuur 8.5 Form to show the types of variables

The first line in the form contains three elements: a label *lblBoolean*, a textbox *txtBoolean* and a label *lblBooleanValue*. Create the similar objects (with corresponding names) for the other types of variables.
Name the command button *cmdOK*.
Set the captions for all objects.
Use the handles to enlarge the form.

When you are designing the form, take a look at the options in the *Layout* menu to align the labels and textboxes. Try using the *Copy* and *Paste* options, for the size of the copies will be exactly the same as the original. This way you'll get the work done much faster compared to changing each object separately.

Add the following code in the subroutine of the [OK] button. Please note the presence of a variable of type Object, that comes in very handy here.

Download free eBooks at bookboon.com

```
Private Sub cmdOK_Click()

  ' Declaration of the different types of variables
  Dim varBoolean As Boolean
  Dim varInteger As Integer
  Dim varLong As Long
  Dim varString As String
  Dim varVariant As Variant

  ' Remember the object in this variable for the error message
  Dim varObject As Object

  ' Go to the label fault, when an error occurs
  On Error GoTo fault

  Set varObject = Me.txtBoolean
  Me.txtBoolean.ForeColor = vbBlack
  varBoolean = Me.txtBoolean
  Me.lblBooleanWaarde.Caption = varBoolean

  Set varObject = Me.txtInteger
  Me.txtInteger.ForeColor = vbBlack
  varInteger = Me.txtInteger
  Me.lblIntegerWaarde.Caption = varInteger

  Set varObject = Me.txtLong
  Me.txtLong.ForeColor = vbBlack
  varLong = Me.txtLong
  Me.lblLongWaarde.Caption = varLong

  Set varObject = Me.txtString
  Me.txtString.ForeColor = vbBlack
  varString = Me.txtString
  Me.lblStringWaarde.Caption = varString

  Set varObject = Me.txtVariant
  Me.txtVariant.ForeColor = vbBlack
  varVariant = Me.txtVariant
  Me.lblVariantWaarde.Caption = varVariant

  Exit Sub

fault:
  ' This variable cannot be set equal to the user input and
  ' will be shown in red
  varObject.ForeColor = vbRed

  ' The remaining program code will be executed
  Resume Next
End Sub
```

Download free eBooks at bookboon.com

Experiment with the form, by typing input in each of the five textboxes (as shown in the figure below) and clicking on [OK].

Variable and Types

Boolean	33	True
Integer	33333	0
Long	12345	12345
String	This is text	This is text
Variant	This is also text	This is also text

OK

Figure 8.6 Filled out form

Struggling to get interviews?

Professional CV consulting & writing assistance from leading job experts in the UK.

Visit site

Take a short-cut to your next job!
Improve your interview success rate by 70%.

TheCVagency
Visit thecvagency.co.uk for more info.

Download free eBooks at bookboon.com

You can use the [Tab]-key to jump from one textbox to another (and use [Shift]+[Tab] to do this in reverse direction). You may adjust the tab order of the objects by setting the *Tab Stop* of each object to *True* or *False* and by changing the numerical value of the propertie *Tab Index*.

Please think about the following issues:

- Why is the value in the textbox *txtInteger* in Figure 8.6 red?
- When is the variable *varBoolean* 'true' and when is the value 'false'?
- Can you type text in the Integer textbox?
- Type texts and numbers in the Variant box.
- Also try typing negative numbers!

Download free eBooks at bookboon.com

9 INTEGRATING CODE

9.1 INTRODUCTION

In this chapter we will show you how you can work with your own programs in worksheets. We will discuss how to open a form or a macro from within a worksheet, and also how you can install your own program into the standard Excel environment.

9.2 START

When you have programmed a form, a sub or a macro, you want to be able to start it from within Excel. Since the spreadsheet program considers (public) subroutines located in modules to be macros, you will find these subroutines in the Macro window.

Open the workbook **VAT Calculation**.
Go to the VBA Editor and create a new module with the following code.

```
Public Sub ShowForm()
  frmVAT.Show
End Sub
```

Switch to the worksheet in Excel.
Go to the *Macro* window (tab *Developer*, *Program code*, *Macros*).

Download free eBooks at bookboon.com

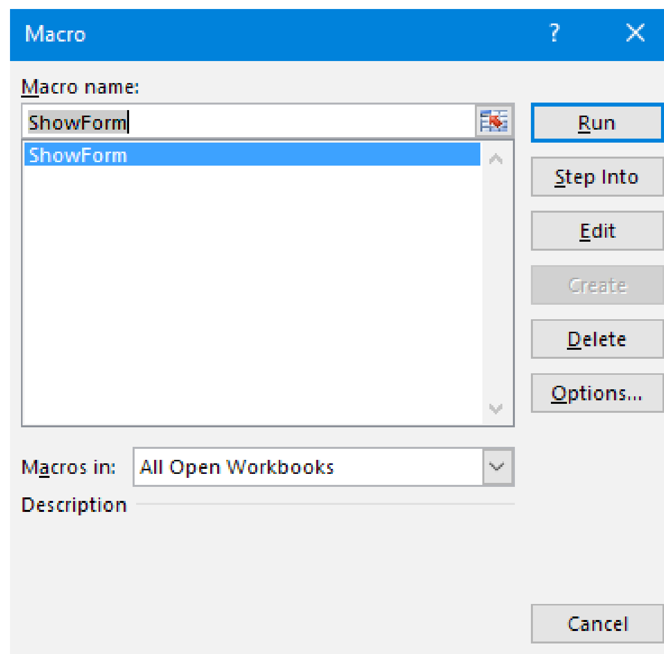


Figure 9.1 Starting a macro

You will find the sub you just created listed as a macro here, and you can start it with the [Run] button. Go ahead.

In the window shown above in Figure 9.1 you can use the [Options…] button to define a shortcut for this macro.

Since Excel sees the sub as a macro you can easily add a button to the Quick Access toolbar to run the macro from there.

Click the small button *Customize* on the right side of the Quick Access toolbar and go to the option *More Commands*.
Select in the window (*Excel Options, Customize the Quick Access Toolbar*) in the left listbox under *Choose commands from:* the option *Macros*.
Add the macro *ShowForm* to the toolbar.

Download free eBooks at bookboon.com

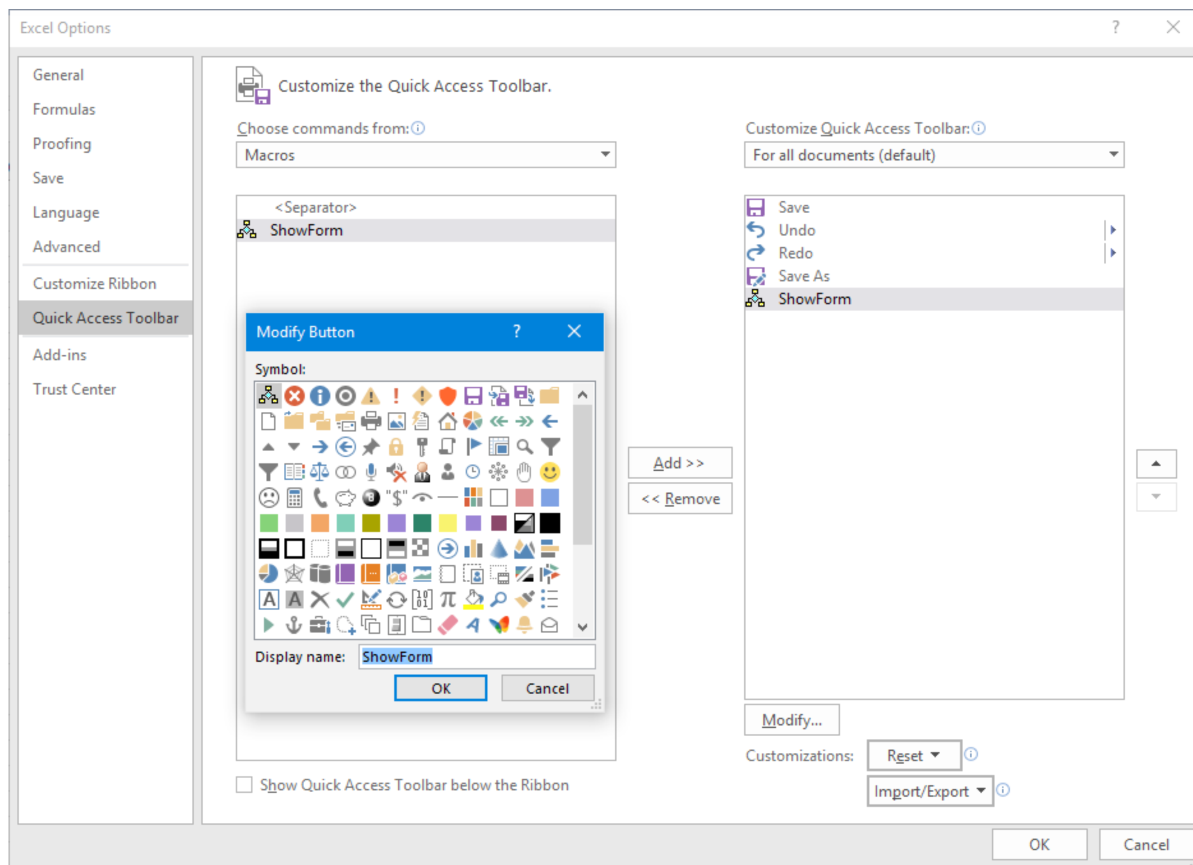


Figure 9.2 Adding a macro button to the Quick Access toolbar

If you like, choose a different icon for the button (so that not all macros and programs will have the same standard icon). You may also modify the *Display name*, this (new) name will be shown in the tooltip that is activated on mouse over.
Run the form with the button on the Quick Access toolbar and perform a calculation.
Close the form and the workbook.

The code for the form has been saved in the worksheet **VAT Calculation**. You might want to use this useful 'program' in other workbooks too.

Start a new workbook. Please note that the macro button is still available on the Quick Access toolbar.
Click the macro button and…the window for the VAT calculation appears.

This method seems very convenient, but you might encounter some problems with it. It only works when the workbook containing the code – the file **VAT Calculation** – is present on your computer and can be found in the same location where it was first saved. In case you move or rename the file, the macro button will not function anymore and an error message will be displayed. Test it if you like.

Download free eBooks at bookboon.com

Some programmers and users have a special file (workbook) containing all sorts of programs, forms and macros, so they can use them in every workbook in Excel by using macro buttons. This obviously works fine when a person has a permanent workspace.

Instead of adding a button to the Quick Access toolbar you can also place a start button on the ribbon. Here too, you will see an error message when the original file containing the code is moved or renamed.

Go to *Excel Options*, *Customize Ribbon* (right clicking the ribbon will get you there as well).
Add a new group to the main tab *View* and name it *Forms*.
Move the macro *ShowForm* from the left list to the new group and pick a nice icon for it. Set the *Display name* for the macro to *VatCalculation*.

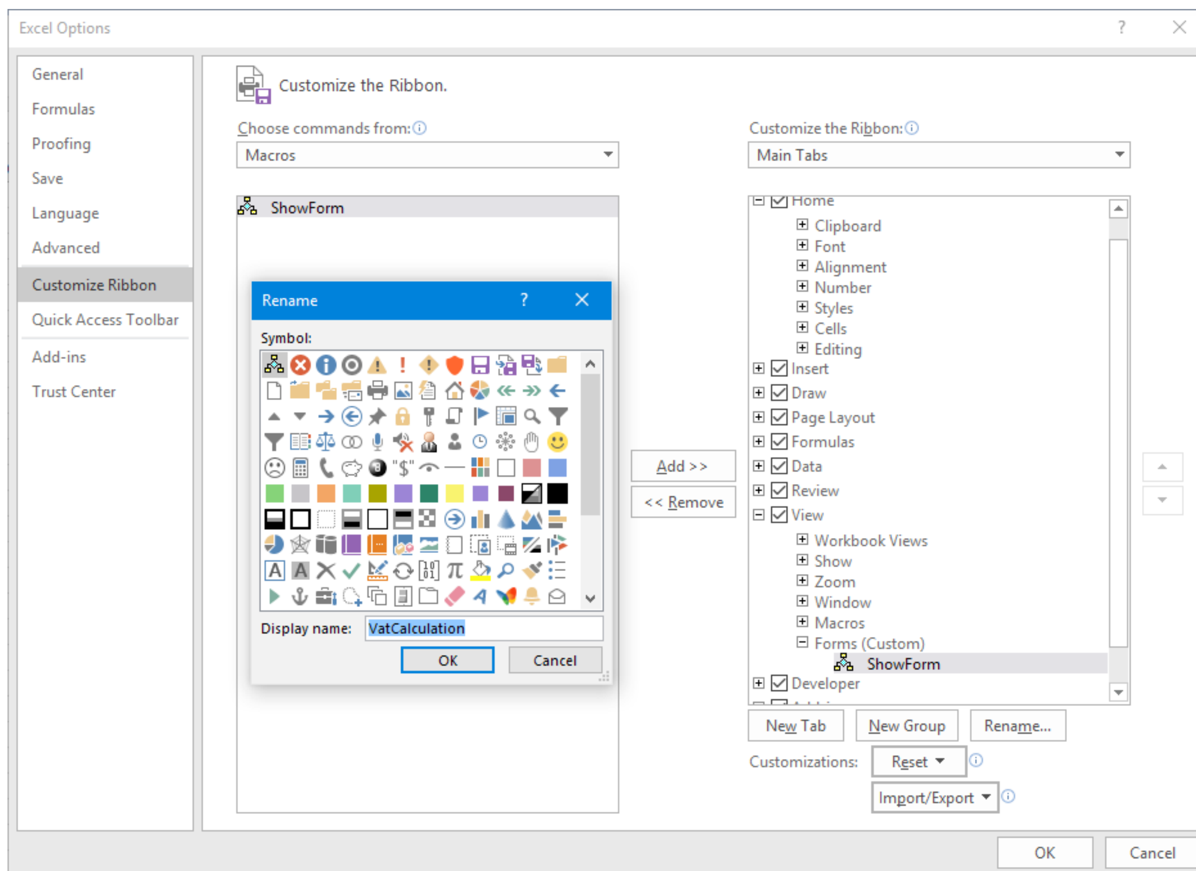


Figure 9.3 Adding a macro button to the ribbon

In the ribbon it will look something like this:

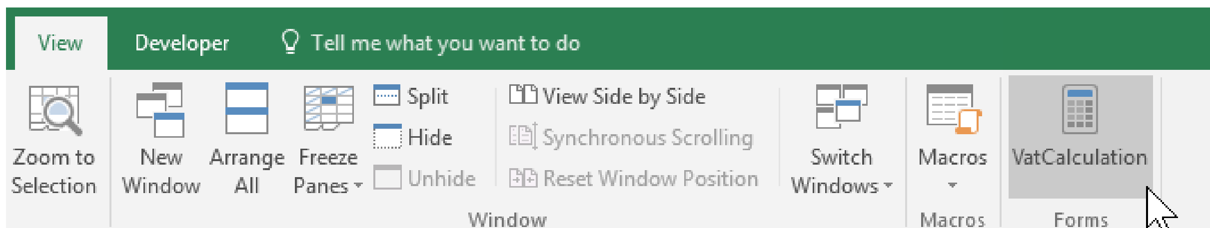


Figure 9.4 Macro button in the ribbon

Open the form with the button in the ribbon.

9.2.1 AUTO_OPEN

A special phenomenon is when a macro is automatically started after opening a workbook. This macro has to be named `Auto_open()` to for Excel to find it. You could of course rename the macro that opens the form to this obligatory name, but then opening the form through other methods is not very convenient. It is better and easier to create a special `Auto_open` sub that calls the `ShowForm` sub.

Go to the code window *Module1* in the VBA Editor and add the following sub.

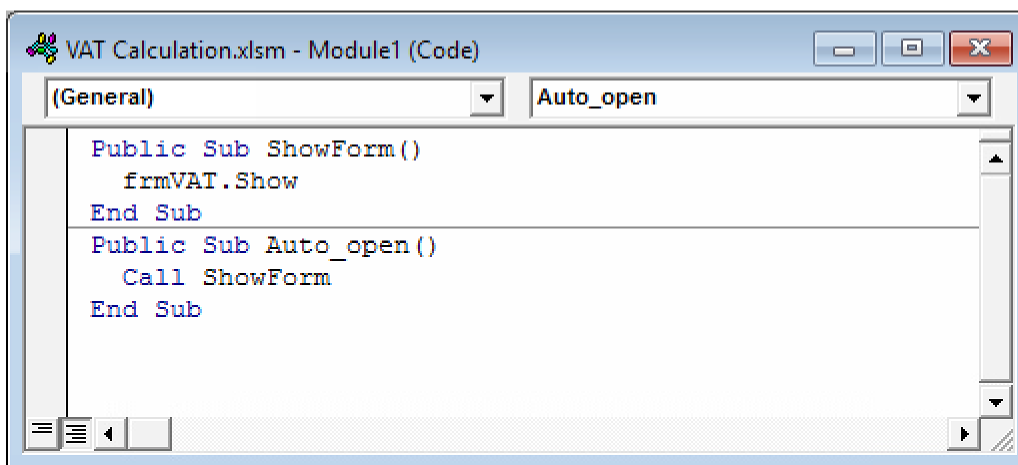


Figure 9.5 Starting a macro automatically

Close and save the workbook.
Open the file **VAT Calculation** again, the form *Calculate VAT* appears immediately.
Close the form and reopen it using the ribbon or the Quick Access toolbar.

9.2.2 WORKSHEET BUTTONS

You can also place a button on the worksheet that opens the form. On the *Developer* tab you'll find a group called *Controls* with elements you can position anywhere on the worksheet.

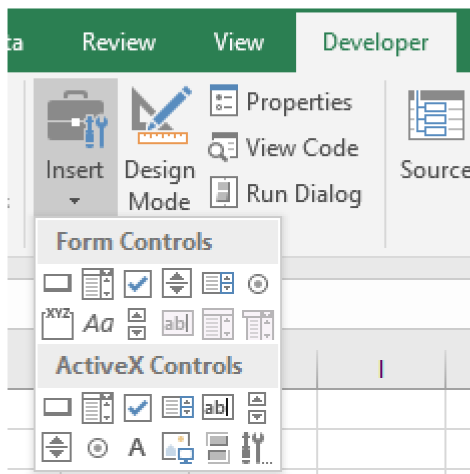


Figure 9.6 Form controls

Insert a command button in the worksheet and assign the macro *ShowForm* to it. Change the caption (on the button).

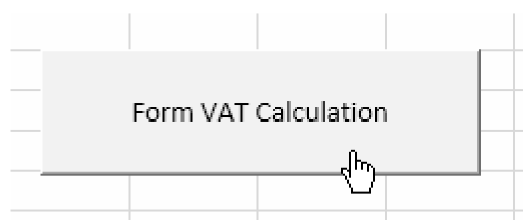


Figure 9.7 A button on the worksheet

Activate a random cell in the worksheet and then hover over the command button. The mouse pointer will change to a small hand (like in a browser), clicking this button now will open the form.

If you want to modify the layout or function of the button, first select it while pressing the [Ctrl]-key. You can also activate the context menu of the button (right click) in order to edit or set the properties.

Not only form controls, also ActiveX controls are available to add to the worksheet (see Figure 9.6). Please be careful with these elements, they can cause all kinds of security issues. Many security applications regard them with much suspicion. It is better to choose the form controls.

Download free eBooks at bookboon.com

9.3 ADD-IN

There is yet another special possibility to integrate code into the Excel environment: namely as an add-in. An add-in allows you to add your own functions to the set of build-in functions of Excel, so that you can use them in a similar way like for example =Sum( ), =Amount( ), =Max( ) etcetera. If you find this interesting, please follow the instructions below carefully and you will discover how this works.

Create, in a new workbook, a new module in the VBA Editor.
Insert the following code for a function called QuantityPrice:

```
Function QuantityPrice(Quantity as Integer, Price as Double)

  Select Case Quantity
    Case Is >= 150
      QuantityPrice = Price * 0.85
    Case Is >= 100
      QuantityPrice = Price * 0.9
    Case Is >= 50
      QuantityPrice = Price * 0.95
    Case Else
      QuantityPrice = Price
  End Select

End Function
```

This code calculates the price of a product depending on what amount of the product is purchased: for every 50 extra items, 5% (more) discount is given. Lets see if the function works properly.

Create the ‘model’ as shown in Figure 9.8.

Download free eBooks at bookboon.com

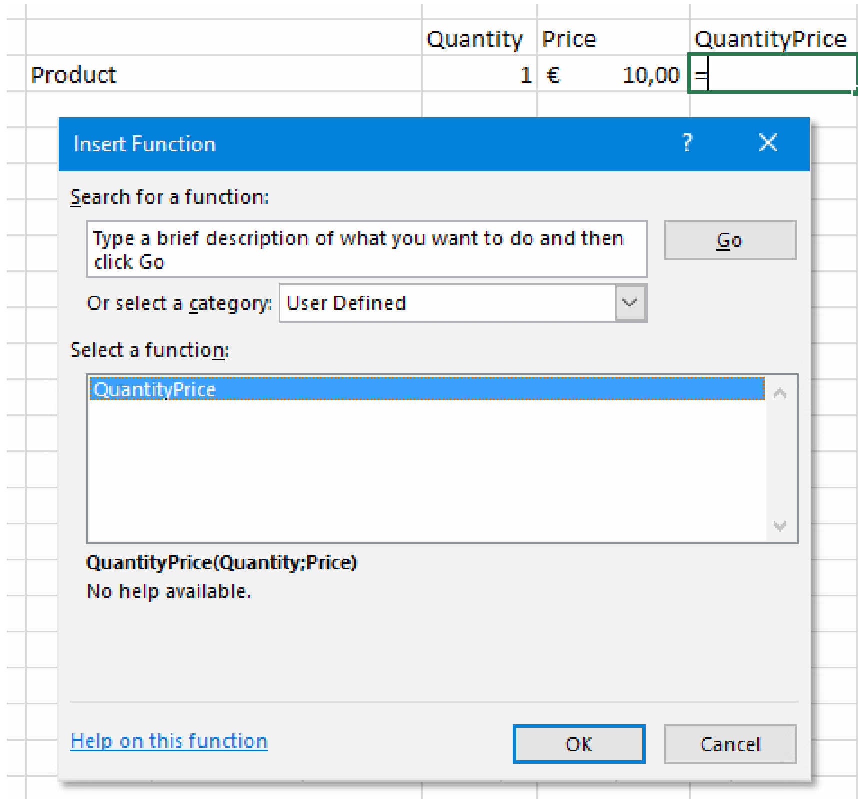


Figure 9.8 A new function

Insert the function as you normally would, for instance by using the Function button in the formula bar. You should now have a category called *User Defined* with the function you have just coded. Select the function and insert the respective references as the *Function Arguments*.

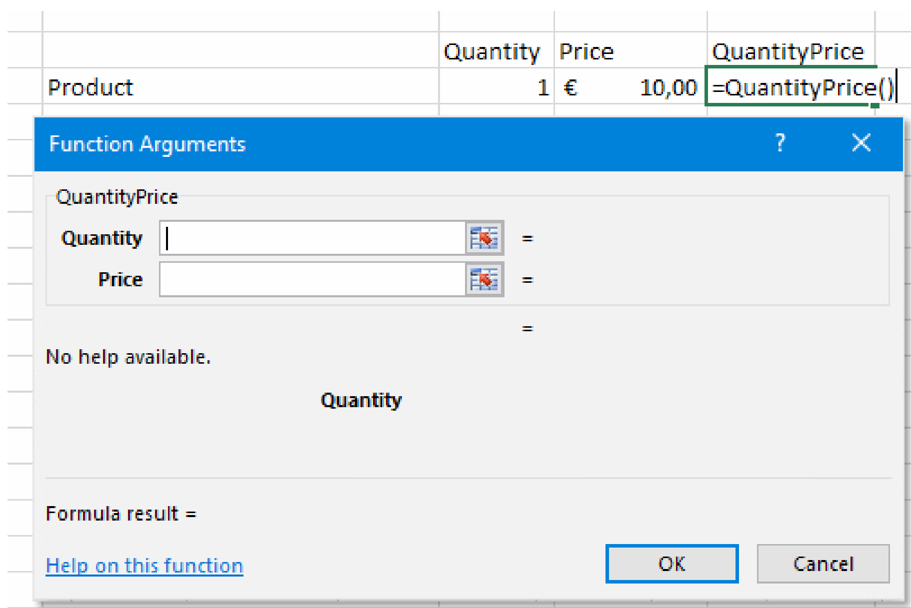


Figure 9.9 Function arguments for your own function

Test different quantities in the spreadsheet and see what the Quantity Price will be.

When it all works properly you can add the function to Excel as an add-in. This will make it available at all times. By the way, you should save the worksheet with the code as well: modification of the function is only possible in this worksheet.

Clear all contents from the cells in the current worksheet.
Save the document as an *Excel Macro-Enabled workbook* named **QuantityPrice**.
Choose *Save as* (again) – or press [F12] – to save the worksheet, now as an add-in. Set the file type at the bottom of the *Save as* window to *Excel Add-in,* the program will automatically open the location where you have to save the **.xlam** file.

e-learning for kids
• The number 1 MOOC for Primary Education
• Free Digital Learning for Children 5-12
• 15 Million Children Reached
About e-Learning for Kids Established in 2004, e-Learning for Kids is a global nonprofit foundation dedicated to fun and free learning on the Internet for children ages 5 - 12 with courses in math, science, language arts, computers, health and environmental skills. Since 2005, more than 15 million children in over 190 countries have benefitted from eLessons provided by EFK! An all-volunteer staff consists of education and e-learning experts and business professionals from around the world committed to making difference. eLearning for Kids is actively seeking funding, volunteers, sponsors and courseware developers; get involved! For more information, please visit www.e-learningforkids.org.

Download free eBooks at bookboon.com
Click on the ad to read more

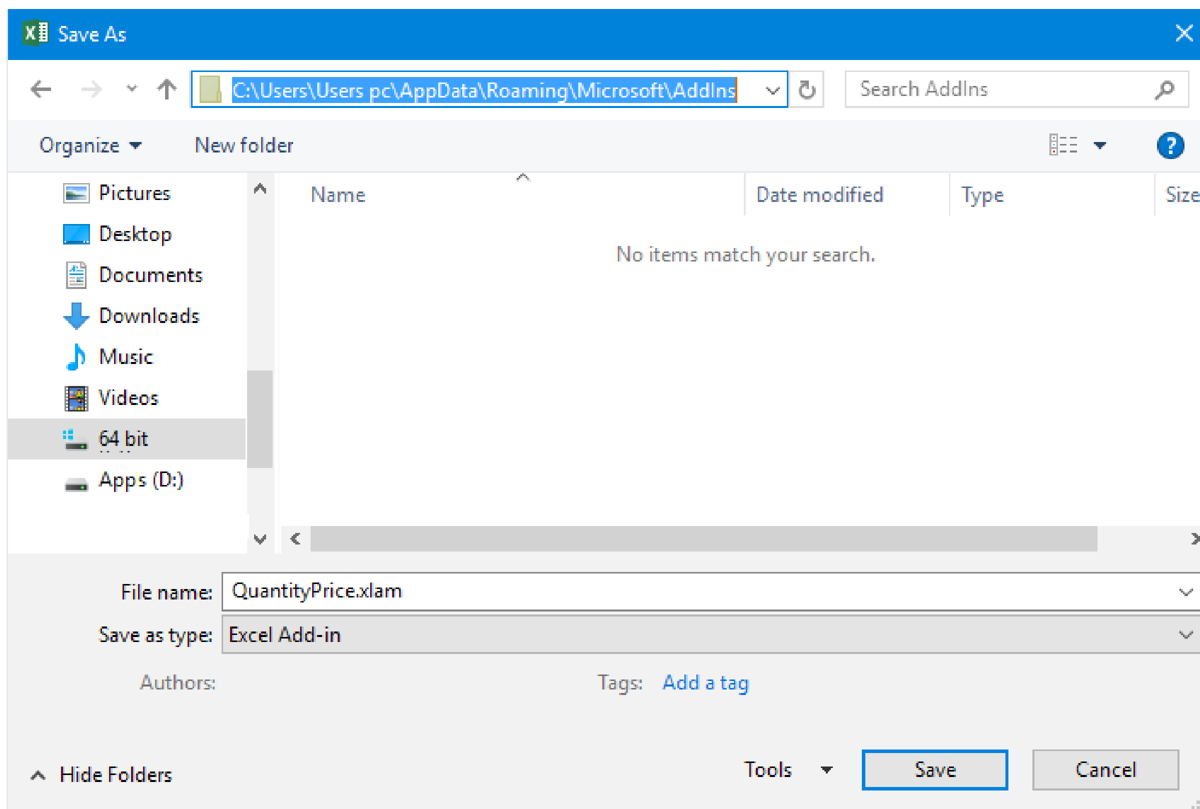


Figure 9.10 Saving as a plug-in

At the top of the window, you can see in which folder your add-ins are saved. Given this information you also know where to delete the function when necessary. The exact location (path and folder) will be displayed when you click once in the Address bar.

Set the name and type as shown in Figure 9.10 and [Save] the file.
Go to the *Developer* tab, group *Add-ins* and click the button *Excel Add-ins*.

Download free eBooks at bookboon.com

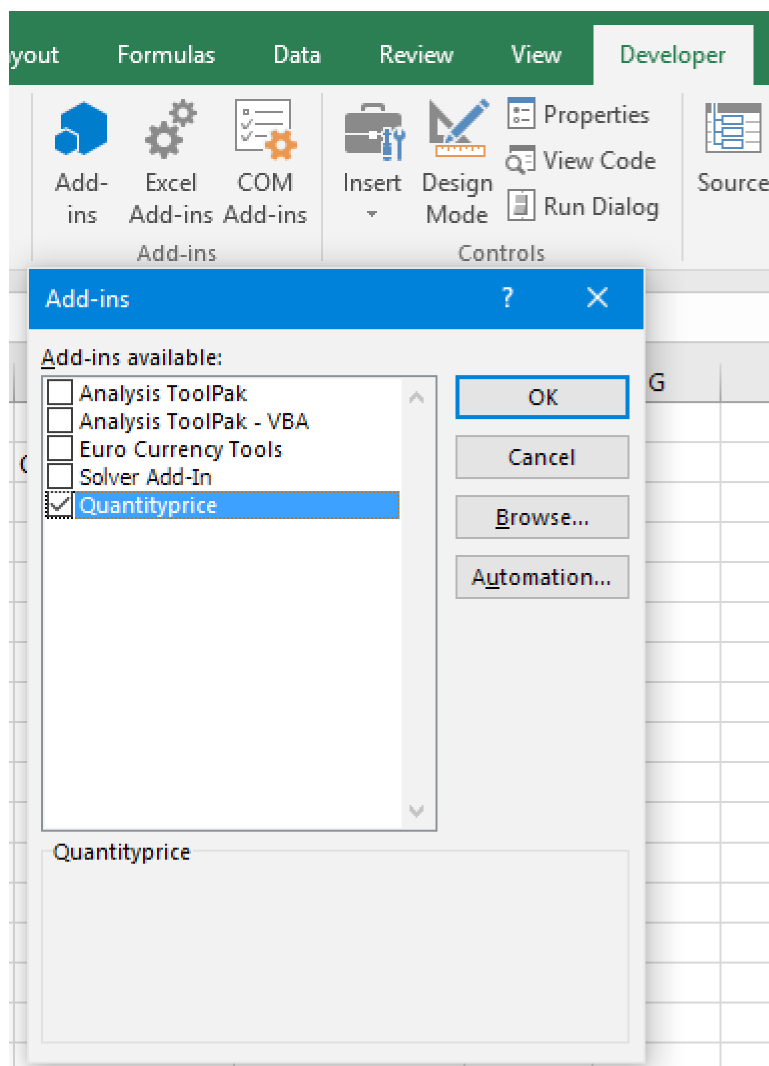


Figure 9.11 Activating your add-in

Activate the add-in (check it and click [OK]). There is a much longer route to activate the plug-in through *Excel Options*. Feel free to examine that path as well. Close all workbooks and start a new one. Check if you can use the function in the new workbook. It should work fine…

10 CASE: AN INVOICE MODEL

10.1 INTRODUCTION

In this chapter we will look more closely at the practical possibilities provided by specific Excel objects such as worksheets, cells and formulas, in combination with VBA. For instance, how do you get certain values from cells or how do you put results into cells? We will no longer discuss every step in these exercises because you are now experienced enough to handle and tackle things in Excel.

10.2 THE INVOICE WORKBOOK

In the example of this chapter we are going to create a workbook to set up invoices for a unique fruit store: it is specialized in gift boxes composed from all kinds of pieces of fruit, which are charged sperately instead of per kilo. The first worksheet in the workbook contains the invoice, or more specifically, the model that holds the invoice lines. The file also contains a catalog of the different products and their list prices.

Create a new workbook and save it as **Invoice** (file type *Excel Macro-Enabled workbook*).
Make sure there are two worksheets in this workbook: one (tab) called *Invoice* and one called *Price List*.
Put labels on row 1 as shown in the figure below (please note the widths of the columns).

Figure 10.1 Tab Invoice

Download free eBooks at bookboon.com

Name cell A4 *Invoiceline* (tab *Formulas*, *Defined Names*). This name will also appear in the name box at the top left.
Assign the *Financial* notation to the columns (D:G) where prices and amounts will appear later.
The worksheet *Price List* should look something like Figure 10.2.

	A	B	C	D
1	**Product**	**Price**	**VAT**	
2	Apple	€ 0,55	5%	
3	Pear	€ 0,65	5%	
4	Lemon	€ 0,45	5%	
5	Orange	€ 0,65	5%	
6	Peach	€ 0,35	5%	
7	Banana	€ 0,20	5%	
8	Apricot	€ 0,30	5%	
9	Fruit Bowl	€ 6,95	20%	
10	Basket	€ 1,95	20%	
11	Wine Box	€ 7,50	20%	
12				

Invoice | Price List

Figure 10.2 Worksheet Price List

Name the cell range A1:A11 on the worksheet *Price List* – where the products are listed – *Product*; the range with the prices should be named *Price* and the range containing the VAT percentages *VAT*. We need these names to look up items from the list later.

10.3 THE INVOICE FORM

Now let's create a form to make selections from the products from the *Price List* tab. Once a product is chosen, the corresponding price and the VAT rate should also be retrieved.

Create a form in the VBA Editor that looks like the one in the figure below. We have used some 'new' controls like a tab control, a spin button (with the arrows on it) and several others, so you will get to know them as well. Please refer to the table on the next pages for the names of all the elements.

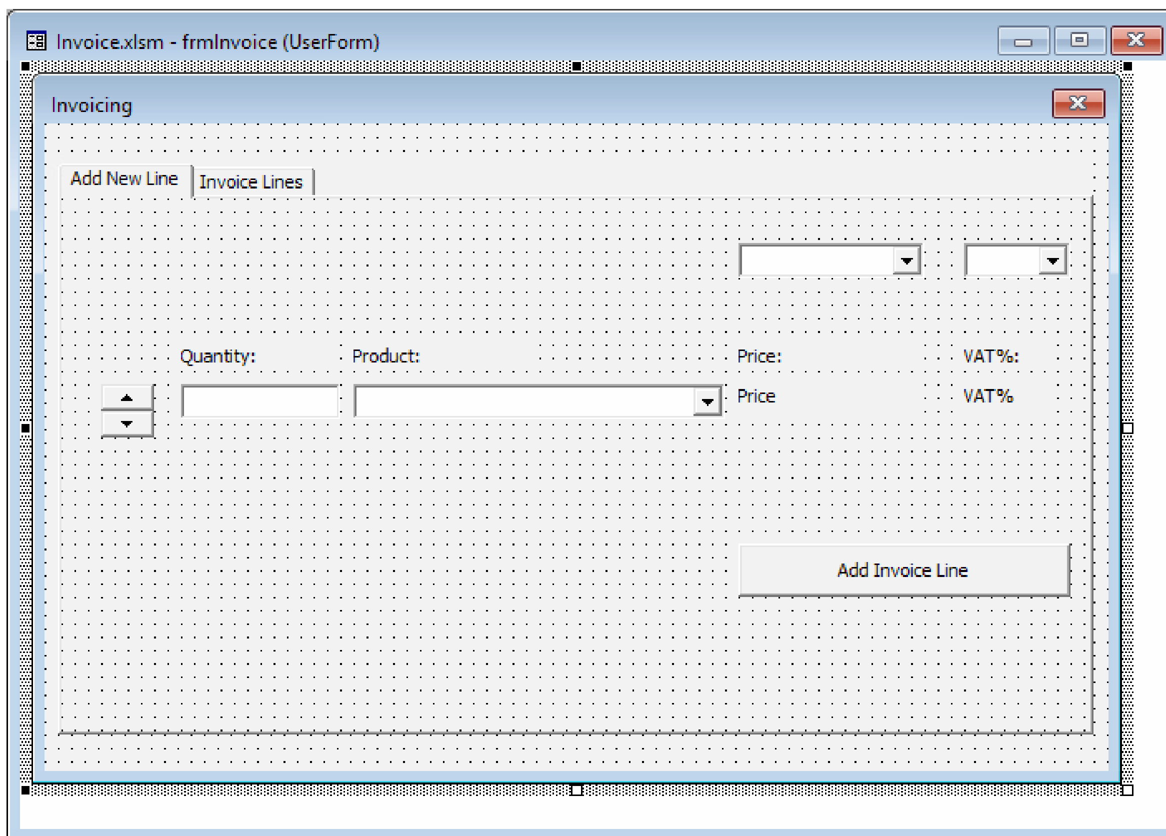


Figure 10.3 The Invoice form

We will get to the tab page *Invoice Lines* later. Please make sure you use the names and captions from the list below; this will prevent problems later on!

Object	Name	Caption
Form	frmInvoice	Invoicing
Tab Control	tabInvoice	
Tab Page Add New Line	pagNewLine	Add New Line
Tab Page Invoicelines	pagLines	Invoice Lines
Spin Button	spnQuantity	
Label Quantity	lblQuantity	Quantity:
Label Product	lblProduct	Product:
Label Price	lblPrice	Price:
Label VAT	lblVAT	VAT%:
Combobox Price	cboPrice	

Object	Name	Caption
Combobox VAT	cboVAT	
Textbox Quantity	txtQuantity	
Combobox Product	cboProduct	
2nd Label Price	lblPrice2	Price
2nd Label VAT	lblVAT2	VAT%
Command Button Add Invoice Line	cmdAdd	Add Invoice Line

Test (the lay-out of) the form, so without any code added yet.
Save the workbook.

10.3.1 THE CODE

You will notice that nothing really happens when you use the buttons or listboxes. So now we will add the code needed for the elements to function as desired.

Close the form and return to the design window in the VBA Editor.
Select the spin button *spnQuantity* and in the *Properties* panel set the value of *Min* to 1, and *Max* to 10 (for larger numbers the spin button is not very useful, then it is easier to type them).
Create a subprocedure to increase or decrease the value in the *txtQuantity* box when you click on the spin button (first try to find a solution yourself; you can find our solution below). Test it in the form.

```
Private Sub spnQuantity_Change()
  Me.txtQuantity = spnQuantity.Value
End Sub
```

The next step after opening the form is to arrange that the comboboxes are filled with the data form the *Price List* worksheet. For this we need the Excel objects `Cell` and `ActiveCell`.

Type the following instructions in the code window of the form.

Download free eBooks at bookboon.com

```
Private Sub UserForm_Initialize()

  ' Make sure the price list sheet is active
  ActiveWorkbook.Worksheets("Price List").Activate

  ' Move to the productr range and skip
to the second cell of the column
  ' the first is only a column header
  ActiveWorkbook.Worksheets("Price List").
Range("Product").Select
  ActiveCell.Offset(1, 0).Activate

  ' Fill the combobox Product with data
  Do While ActiveCell.Text <> ""
    Me.cboProduct.AddItem ActiveCell.Value
    ActiveCell.Offset(1, 0).Activate
  Loop

  ' Fill the combobox Price with data
  ActiveWorkbook.Worksheets("Price List").Range("Price").Select
  ActiveCell.Offset(1, 0).Activate
  Do While ActiveCell.Text <> ""
    Me.cboPrice.AddItem ActiveCell.Value
    ActiveCell.Offset(1, 0).Activate
  Loop

  ' Fill the combobox VAT with data
  ActiveWorkbook.Worksheets("Price List").Range("VAT").Select
  ActiveCell.Offset(1, 0).Activate
  Do While ActiveCell.Text <> ""
    Me.cboVAT.AddItem ActiveCell.Value
    ActiveCell.Offset(1, 0).Activate
  Loop

  Me.txtQuantity = me.spnQuantity.Value

End Sub
```

When something goes wrong, please use the *Debug* options to run through the procedure step by step to find (and correct) any errors in the code.

We now have to modify the combobox `cboProduct` so that when you choose a product from the list, you will get to see the corresponding price and VAT-percentage in the form. You can do this with the following subprocedure.

Download free eBooks at bookboon.com

```
Private Sub cboProduct_Change()
  Me.cboPrice.ListIndex = Me.cboProduct.ListIndex
  Me.cboVAT.ListIndex = Me.cboProduct.ListIndex

  Me.lblPrice2 = Format(Me.cboPrice, "€ #,##0.00")

  Me.lblVAT2 = Format(Me.cboVAT, "0%")

End Sub
```

Add the instructions above at the appropriate position in the code window. Study the code carefully to 'predict' what it will do.
Test the form, try choosing a different product from the list and see what happens.

The comboboxes `cboPrice` and `cboVAT` with the 'actual' corresponding data of the products, are of no use to the user. They have to be present to make the form work. Instead of the combo's we could have used an *array variable* (which we will do later on), but in this case the comboboxes are easier to use.

Make the comboboxes invisible for the user (set the *Visible* property to *False*).

10.3.2 LIST WITH INVOICE LINES

The chosen product, the quantity and the price have to be remembered in order to write the data on the invoice all at once. We will first modify the tab page *Invoice Lines* for this by adding a listbox and command button.

A listbox has several properties to configure the columns in the list, such as the *ColumnCount* property that determines the number of columns in the listbox. The number of rows is of less importance, they can be changed during execution. You can choose the height of the listbox as you like. If not all invoice lines fit in, the listbox will show a scroll bar. The *ColumnHeads* property is used to set whether the names of the columns should be visible or not. The setting *ColumnWidths* determines the widths of the columns in the listbox. Preferably all data in the columns should be entirely visible.

Modify the tab page *pagInvoice* to match Figure 10.4. Name the listbox *lstInvoice* and the command button *cmdOK*.

Download free eBooks at bookboon.com

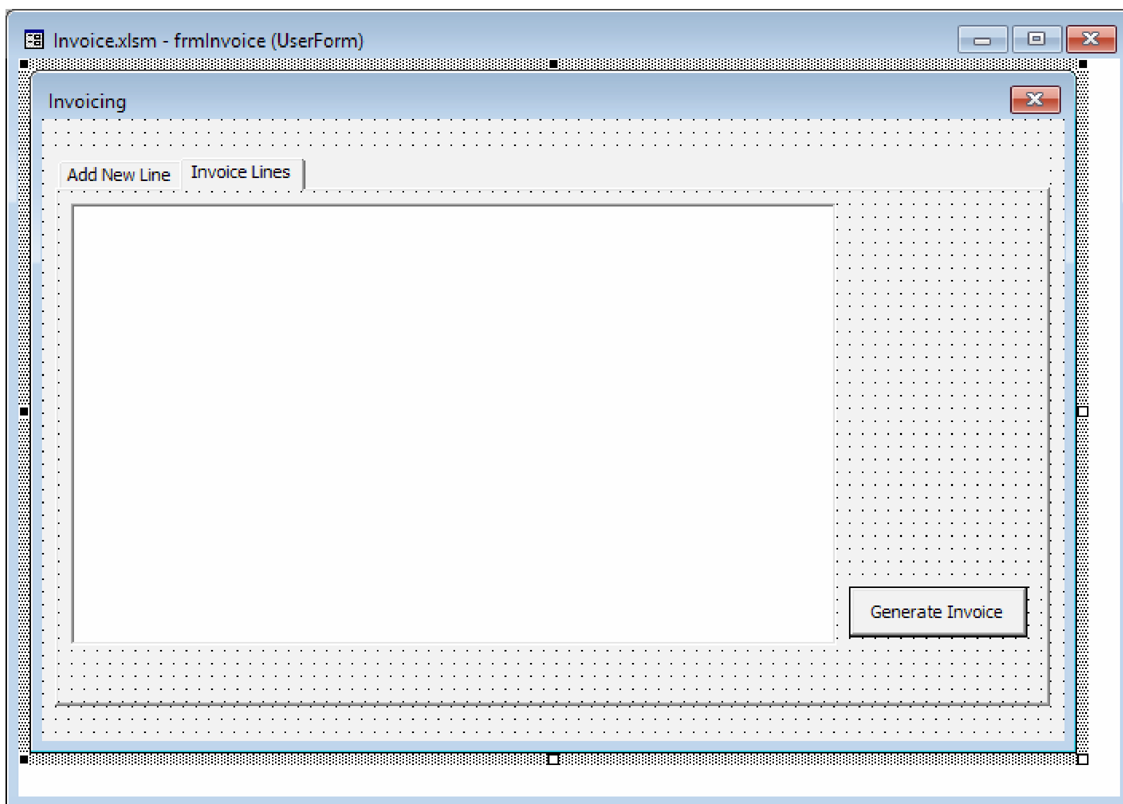


Figure 10.4 Tab page Invoice Lines

Set the number of columns (*ColumnCount*) and their widths (*ColumnWidths*) of the listbox as shown in the figure below (for unknown reasons Excel will modify the input slightly).

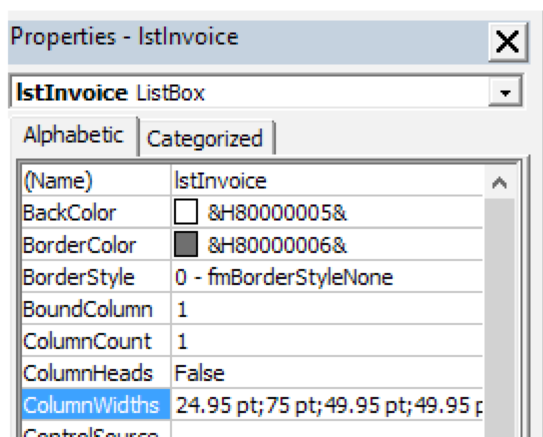


Figure 10.5 Column settings of the list

A listbox can be filled with data using an *array*, which is basically a *matrix* of variables. This matrix can be one, two- or n-dimensional. You will first have to fill the array with data and then link it to the listbox. Please note: in VBA the first row and column of the matrix are referred to as row 0 and column 0! You'll see it in the next paragraph.

10.3.3 CREATING INVOICE LINES

In order to create the list of products on the invoice, we have to add code to the *cmdAdd* button on the first tab page of the form to transfer the chosen product, quantity, price and VAT to the listbox on the second tab page. When the [Add Invoice Line] button is clicked again, a next line has to be added to the listbox of course.

Create the subprocedure shown on the next page. Please read the included comments carefully to see what the code actually does.

FACTCARDS
Are you working in academia, research or science? And have you ever thought about working and moving to the Netherlands?
Arriving 33
Living 50
Studying 51
Working 101
Research 50
Factcards.nl offers all the **information** that you need if you wish to proceed your **career** in the **Netherlands**.
The information is ordered in the categories arriving, living, studying, working and research in the Netherlands and it is freely and easily accessible from your smartphone or desktop.
VISIT FACTCARDS.NL
Click on the ad to read more
Download free eBooks at bookboon.com

```
Private Sub cmdAdd_Click()

  Dim nLines As Integer
  Dim i, j As Integer

  ' How many lines are there in the listbox?
  nLines = Me.lstInvoice.ListCount

  ' Define a (two-dimensional) array.
  ' We have to use ReDim instead of Dim, because the size
  ' of the array depends on the variable nLines.
  ' The 3 means that there are four columns: 0, 1, 2 and 3.
  ReDim invLine(nLines, 3)

  ' Fill the last row of the array with the contents of the
  ' elements on the Add New Line tab page
  invLine(nLines, 0) = Me.txtQuantity
  invLine(nLines, 1) = Me.cboProduct
  invLine(nLines, 2) = Me.cboPrice
  invLine(nLines, 3) = Me.cboVAT

  ' The existing content of the listbox has to be
  ' retransferred to the array!
  For i = 0 To nLines - 1
    For j = 0 To 3
      invLine(i, j) = Me.lstInvoice.Column(j, i)
    Next j
  Next i

  ' The content of the array is (re-)linked to the listbox
  Me.lstInvoice.List = invLine

End Sub
```

Test the command button *cmdAdd* a few times and look at the results on the *Invoice Lines* tab. If you like, you can use the debugging options to see what happens step by step.

10.4 CREATING THE INVOICE

Now the invoice itself has to be created. Each time the [Generate Invoice] button (= *cmdOK*) on the *Invoice Lines* tab page is clicked, a copy of the empty *Invoice* worksheet is made and the data in the listbox is transferred to that sheet.

Create the subprocedure `cmdOK_Click()` as shown below (do not forget to read and study the comments).

```
Private Sub cmdOK_Click()

  ' Creating the invoice by coping an empty invoice sheet
  Sheets("Invoice").Select
  Sheets("Invoice").Copy After:=Sheets(2)

  ' Activate the new invoice sheet
  ActiveWorkbook.Worksheets(3).Activate

  ' Go to the correct location on the invoice sheet
  ActiveWorkbook.Worksheets(3).Range("Invoiceline").Select

  ' Fill the sheet with data from the listbox
  For x = 0 To Me.lstInvoice.ListCount - 1
    With ActiveCell
      .Value = x + 1
      .Offset(0, 1) = Me.lstInvoice.Column(0, x)
      .Offset(0, 2) = Me.lstInvoice.Column(1, x)
      .Offset(0, 3) = Format(Me.lstInvoice.Column(2, x), "€ #,##0.00")
      .Offset(0, 4) = Format(Me.lstInvoice.Column(3, x), "0%")

      ' Go to the next line
      .Offset(1, 0).Activate
    End With
  Next x
End Sub
```

Test the form and make sure everything works properly.
Save the workbook.

10.4.1 FORMULAS IN VBA

The invoice also requires some totals. So we have to add formulas at the appropriate positions on the new worksheet. Formulas look slightly different in VBA code compared to the formulas in the Excel worksheet. To give you an idea of how formulas are built in VBA, we will start by creating a macro that puts a formula in a cell.

Download free eBooks at bookboon.com

Type two numbers in two consecutive cells in a new worksheet as shown in the figure below.
Start recording a new macro and calculate the sum of the two numbers using the *AutoSum* button.
Now stop recording.

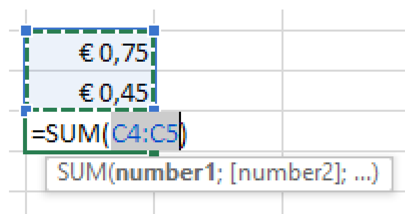


Figure 10.6 Recording a formula in a macro

Study the macro code in the VBA Editor.

```
ActiveCell.FormulaR1C1 = "=SUM(R[-2]C:R[-1]C)"
```

This formula can be interpreted as: 'calculate the sum of the values in the range that starts with the cell 2 rows above the current cell and ends with the cell right above the current one'. VBA uses a string to represent the formula and then writes this string into the active cell.

In the formula, Excel uses a *relative reference* to the cells in the worksheet, even though we have not set this option at the start of the recording. In case you did, the code line starts with `Selection` instead of `ActiveCell`, but what follows is exactly the same.

Building the formula using `ActiveCell` is a useful method to calculate the end total of the invoice later. After all it is not clear in advance how many lines the invoice will have.

We can 'abuse' the way VBA represents formulas'' to create our own strings that we can put together to obtain the formula we want. It'll probably take you some trial and error to concatenate the right characters and stings. We will now use this method to calculate the total amount at the end of each invoice line (both excluding and including VAT).

Close the new workbook (without saving), and go back to the **Invoice** workbook. Add the following code in the `cmdOK_Click`() sub, just before the instruction to go to the next line (so within the loop of the counter variable x):

Download free eBooks at bookboon.com

```
.Offset(0, 5).Formula = "=B" & (x + 4) & "*D" & (x + 4)
.Offset(0, 6).Formula = "=F" & (x + 4) & "*(1+E" & (x + 4) & ")"
```

Study carefully how the string for the formula is built with these instructions. Run the subprocedure and examine the formulas that are created in the (added) copy of the *Invoice* worksheet.

As an alternative you could also use the following code:

```
.Offset(0, 5).Formula = "=RC[-4]:RC[-1]"
.Offset(0, 6).Formula = "=RC[-5]:RC[-2]"
```

Now let's look at the total amount of the invoice. When all invoice lines are added, this total has to be at the bottom of the invoice! Though we do not know beforehand how many lines will be involved in the addition – because it depends on the number of invoice lines that the user creates – we know, however, the end value of the counter x after finishing the `For-Next`-loop. Using that value you can build the appropriate (relative) formula.

Insert the following code at the correct position in the OK-sub (right after the loop).

```
' Calculate the total amount of the invoice
ActiveCell.Offset(1, 6).Activate
ActiveCell.Formula = "=SUM(R[-" & x + 1 & "]C:R[-2]C)"
```

Test the whole program again and study the formulas, for instance, when you have specified 3 or 4 lines on the invoice.
Delete all the (temporary) invoice sheets that are created during the testing.

10.4.2 INVOICE NUMBERS

Say you want to give each invoice its own invoice number. We first have to indicate where on the empty invoice form we want to display this number. We choose cell A3 and give this range (of one cell) the name *Invoicenumber*. On the *Price List* tab we type in cell E2 (under an explanatory text in cell E1) the invoice number we want to start with and name this cell *Number*. We will use this cell each time we make a new invoice to save the last used invoice number.

Download free eBooks at bookboon.com

On the *Invoice* tab give cell A3 the defined name *Invoicenumber*.
Enter the current invoice number – for instance the year in four digits followed by a serial number of three digits: 2019001 for the first invoice in 2019 – in cell E2 of the *Price List* tab. Name that cell *Number*. Type a label in cell E1 above it: *Last used invoice number*.
Below you see the required code to put the invoice numbers on the invoices. Insert the code at the correct position.

```
' Add Invoice number
ActiveWorkbook.Worksheets(3).Range("Invoicenumber") _
= ActiveWorkbook.Worksheets(2).Range("Number")

' Increase the Invoice number on the Price List tab
ActiveWorkbook.Worksheets(2).Range("Number") _
= ActiveWorkbook.Worksheets(2).Range("Number") + 1
```

Please note the syntax of typing the long lines of code above: if you want to spread program code over multiple lines in the code window, you can type a space and underscore at the end of the first part of the code line and continue on the next line (with indentation if you like). VBA will consider this to be one code line.

10.5 SECURITY

It is, of course, preferable that not just anybody can change the invoice number – and maybe the prices as well – on the *Price List* tab. You can prevent this by securing the tab *Price List* with a password. The question is: how does VBA deal with this?

Set a password for the *Price List* worksheet (not the complete workbook) on the *Review* tab. Use as a password the (rather unsafe) string 'secret' in lower case characters.
Run the program. At the end when the program wants to create an invoice, an error message appears.

Download free eBooks at bookboon.com

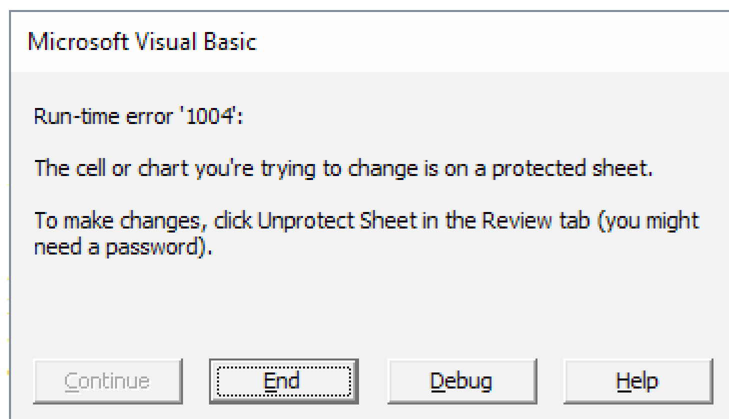


Figure 10.7 Run-time error message

You have to insert code so that the program first deactivates the protection before the value of the last used invoice number in the cell *Number* can be updated. After that, the security measure has to be reinstated.

Insert the two following lines of code at the correct position in the program.

```
ActiveWorkbook.Worksheets(2).Unprotect Password:="secret"

ActiveWorkbook.Worksheets(2).Protect Password:="secret"
```

Test the program again.
Check if the program has increased the invoice number and if the protection has been reinstated.

10.6 USER INTERFACE

Finally, at the end of this module about the invoice use-case, we add some elements to ensure that users can work with your application in a pleasant way. For example, with the use of command buttons in the interface, you can guide and explain the use of the program.

Create some space under the tab control in the form (see Figure 10.8 below) and add a button [Cancel] to close the form without an invoice being created. Insert the correct code for the button (remember what it was?).

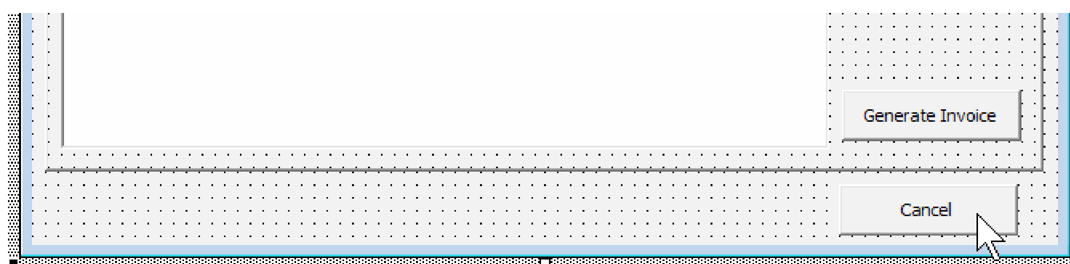


Figure 10.8 Cancel button

Go to the *Price List* tab and unprotect the sheet.
Add a command button (somewhere below the last used invoice number) for the user to open the form to create an invoice. Activate the protection again and test the working of the two buttons you just added.

	A	B	C	D	E	F	G
1	**Product**	**Price**	**VAT**		**Last used invoice number**		
2	Apple	€ 0,55	5%		2019002		
3	Pear	€ 0,65	5%				
4	Lemon	€ 0,45	5%				
5	Orange	€ 0,65	5%		Create Invoice		
6	Peach	€ 0,35	5%				
7	Banana	€ 0,20	5%				
8	Apricot	€ 0,30	5%				
9	Fruit Bowl	€ 6,95	20%				
10	Basket	€ 1,95	20%				
11	Wine Box	€ 7,50	20%				

Figure 10.9 Button on the sheet to create a (new) invoice

10.7 EXTRA EXERCISE

When working with an invoice program, there will be users that only want to create invoices while others have to set the prices and manage the articles. And maybe another department has to provide the numbers for the invoices. In many situations it is undesirable that all users have access to everything. You have already seen how to setup basic protection within your program, but the possibilities are not as extensive as for example in securing a company network or a database program like Access. In such environments, each user normally has his own password and own rights.

However, within the limited options there are still some tricks you can use to prevent users from changing data (deliberately or accidentally).

Create a command button on the *Price List* sheet with the caption *Edit Price List* (as shown in Figure 10.10).
Add the following code to the button.

```
Sub cmdEdit()
  If InputBox("Password") = "excel" Then
    MsgBox "Good guess, you may continue!"
    ActiveWorkbook.Worksheets(2).Unprotect Password:="secret"
    ActiveSheet.ShowDataForm
    ActiveWorkbook.Worksheets(2).Protect Password:="secret"
  Else
    MsgBox "Access denied", vbExclamation
  End If
End Sub
```

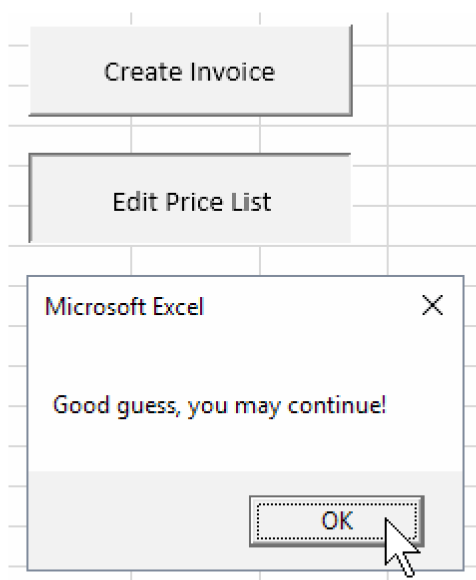


Figure 10.10 Password protected Editing

Test the program and see what the code does (it handles the password and shows a standard form for the data table).

Please think about the following issues:

- What password protects changing the price list?
- After closing the data form, can you still modify the data in the worksheet?
- How can you sort the product list?

Save and close the workbook.

Download free eBooks at bookboon.com

11 CHARTS IN VBA

11.1 INTRODUCTION

VBA in Excel offers many possibilities to create, edit or handle charts. In this last chapter, we will look at a simplified example that uses a chart to help you get started with this subject. We'll design a chart for a rental organization that has bikes, e-bikes and scooters. The chart shows an overview of the rentals of the past week to help the organisation to decide in what products to invest. We will create the chart solely with the use of VBA-code to show you how to handle objects like these.

11.2 THE WORKBOOK

The rental company keeps track in a spreadsheet of how many bikes, e-bikes and scooters are rented in a single week. In the workbook there are three buttons, each displays the results of one of the categories in a chart to the user. The old chart is deleted before a new one is shown. This makes the model we use here fairly simple.

Create a new workbook and save it as **Rental Charts** (*Macro-Enabled*). The workbook must contain three worksheets with repsective tab labels *Bikes*, *E-Bikes* and *Scooters*.
Select all three worksheets (use the [Shift] key) and enter for them the following data.

	A	B	C	D	E
1		Total			
2	Sunday				
3	Monday				
4	Tuesday				
5	Wednesday				
6	Thursday				
7	Friday				
8	Saturday				
9					
10					
11					
12					
13					
14					

Bikes | E-bikes | Scooters

Figure 11.1 Filling a worksheet group

Download free eBooks at bookboon.com

Enter random numbers (of rentals) between 1 and 10 in the cells B2:B8 on each sheet (or use the formula =RANDBETWEEN(1;10)).
Select the cells A1:B8 on the first worksheet and name it *Bikes*. Name the corresponding ranges on the second and third sheet *Ebikes* and *Scooters*.

The use of a hyphen in the name of a range is not allowed, this is why you have to name the range *Ebikes* (without hyphen).

11.2.1 THE CODE

In this section you will find the code you need to create a chart from the data. We will add buttons later to activate the code.

Open the VBA Editor and add a module named *modChart*.
Type the following instructions and procedures.

Brain power

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.
Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.
By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations.
Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

SKF

Click on the ad to read more

```
Option Explicit

Sub Bikes()
  On Error Resume Next
  Dim ChartRange As Range
  Dim Chart1 As Chart

  Charts(1).Delete
  Worksheets("Bikes").Activate
  Range("Bikes").Select
  Set ChartRange = Selection
  Set Chart1 = Charts.Add
  Call Wizard(ChartRange, Chart1, "Bikes")
  Charts(1).Move after:=Sheets(Sheets.Count)
End Sub

Private Sub Wizard(ChartRange As Range, ChartVar As Object, _
    ChartTitle As String)
  ChartVar.ChartWizard _
  Source:=ChartRange, Gallery:=xl3DColumn, Format:=6, _
  PlotBy:=xlColumns, CategoryLabels:=1, SeriesLabels:=1, _
  HasLegend:=False, Title:=ChartTitle
End Sub
```

11.2.2 EXPLANATION

The explanation of the code:

- After declaring the variables the instruction `Charts(1).Delete` deletes a previously added chart first (if there is one, if not this still works because of the `On Error Resume Next` instruction).
- Then the first worksheet is activated and the range *Bikes* is selected. The variable `ChartRange` is set to the values of the range selection.
- Using `Set Chart1 = Charts.Add` a chart is created and assigned to the variable `Chart1`.

Please note in the subprocedure `Bikes()` the special types of variables you can use: `Range` en `Chart`. Search for more information on these types of variables in Help or on internet.

- The instruction `Call Wizard(ChartRange, Chart1, "Bike rentals")` calls the subprocedure `Wizard` and transfers the range of the chart (`ChartRange`), the chart itself (`Chart1`) and the chart title ("Bike rentals") to the sub.
- `Charts(1).Move after:=Sheets(Sheets.Count)` will add the chart on a new sheet after the third worksheet.
- The sub `Wizard` contains the settings of the chart. Besides setting range, chart and title this sub sets the chart type to 3D columns, disables the legend and determines that the first column contains the category labels and the first row the labels of the series.

Add similar code for the subprocedures `E-bikes` and `Scooters`. Make sure the right sheets and ranges will be selected in the procedures. Be carefull: the hyphen of "E-bikes" could cause some problems.
Don't forget to add comments to the source code where necessary.

11.3 USE OF THE VBA-APP

The VBA-part of this small application is now finished. By now you know several ways to run the VBA-code from the interface; the method we demonstrate here is using buttons in a customized part of the ribbon.

Create a new (custom) tab in the ribbon to the right of *Home* and name it *Rental Chart* with one group named *Categories*. Add the three subs as macros and give them adequate names. You may also add an appropriate icon to the macros (the choice is quite limited here).

Download free eBooks at bookboon.com

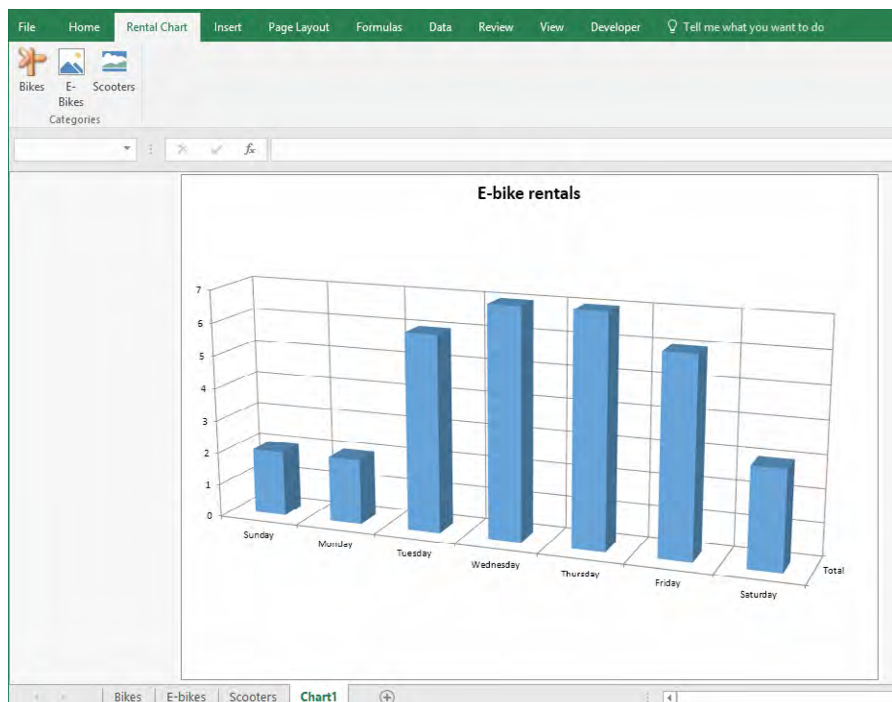


Figure 11.2 Buttons on a custom tab in the ribbon

Test the program: try all buttons of the ribbon tab *Rental Chart*.

A disadvantage of customizing the ribbon is that the modification, e.g. the custom tab *Rental Chart*, remains visible after the file **Rental Charts** has been closed. You have encountered this 'problem' before. An alternative method could be to place a command button on all three sheets that then generates the corresponding chart on that sheet.

When executing the program like this, each time a new chart is created a dialog box will appear to inquire what to do with the sheet holding the previous chart (Cancel will keep the old one).

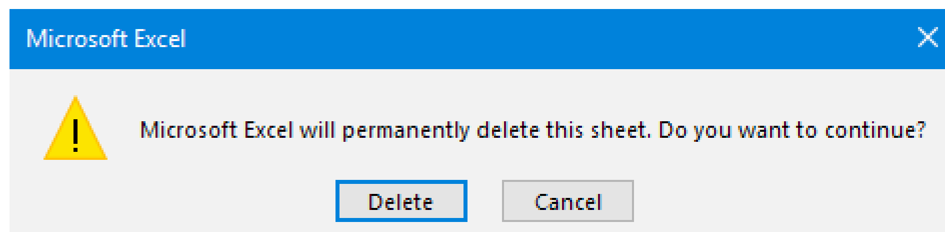


Figure 11.3 Forced to choose what to do with the previous chart

You may suppress these alerts by inserting the following code around the Delete-instruction.

```
Application.DisplayAlerts = False
Charts(1).Delete
Application.DisplayAlerts = True
```

This suppresses only the message shown in Figure 11.3 when deleting the graph; all further messages will appear as necessary (because they are re-enabled in the last line). If you want to suppress the alert for all graphs, you will have to add the two lines of code in all the subs.

Edit the code (in case you want to suppress the alerts) and test the program again.
Looks good, right?
Save and close the workbook.

Download free eBooks at bookboon.com