Excel VBA: Introduction

Excel VBA programming - Part 1

Harun Kaplan

HARUN KAPLAN

EXCEL VBA: INTRODUCTION

EXCEL VBA PROGRAMMING - PART 1

Download free eBooks at bookboon.com

Excel VBA: Introduction: Excel VBA programming - Part 1
1st edition
© 2018 Harun Kaplan & bookboon.com
ISBN 978-87-403-2404-4

CONTENTS

	Introduction	**6**
1	**Visual Basic for Application – VBA**	**8**
2	**Developer Environment**	**9**
2.1	VBA-Editor Environment	10
3	**"Learning by recording" Record of macro**	**18**
3.1	Structure of a procedure	18
3.2	Delete, export or import of Macros / Procedure	24
3.3	Start a macro	25
4	**Script concept by VBA**	**30**
4.1	Procedure, Module, VBA-Code	30
4.2	Variable, constant and date type	36
4.3	Arithmetic-, comparison- und logical Operators	50

Download free eBooks at bookboon.com

Click on the ad to read more

5 Errors in VBA 54
5.1 Errors examples 54
5.2 Handling errors 55
5.3 Toolbar Debug 56

6 Program sequence, branch und loops 59
6.1 Decision structure, If query, Branches and returns 59
6.2 Loops 65
6.3 Branch statement 73

7 Communication with Excel 77
7.1 Message window 78
7.2 Break up or concatenate multiline text lines 82
7.3 Symbol in Message window 83
7.4 Input window 85

Bibliography 88

Download free eBooks at bookboon.com

INTRODUCTION

This book begins with an introduction to the basics of the editor environment and language concept of VBA programming. The variety of examples that result from the practice illustrates the elements of the VBA language to the user with increasing complexity.

The target group of this book is therefore suitable for both beginners and advanced users, such as users who are up to the advanced user provided.

These include users who:

- want to work extensively with Excel.
- have no programming knowledge and want to learn VBA programming.
- have already started programming in VBA or other languages.
- want to expand their VBA skills.

There are already different books and online guides available. So you ask yourself “What’s so special about this book?”.

The peculiarity of the book is that it finds its origin in practice and this can be used excellently as a reference book. The storage locations are not only local places like the hard disk, but also servers.

In martial arts you learn many ways to fend off an attack. It is important, in an emergency, not to think about which technique is to be used, but to carry out a defense for each attack.

In VBA there are also plenty of ways to solve a problem, so you have at least one way to reach your goal.

This book series consists of four parts:

⇨ Excel VBA - Introduction
⇨ Excel VBA – Working with Excel Elements
⇨ Excel VBA – Working with Excel Functions & Data
⇨ Excel VBA – Working with ToolBar Controls

Harun Kaplan

Download free eBooks at bookboon.com

For my Family:
Tülay
Yasin, Sueda, Melik

Download free eBooks at bookboon.com

1 VISUAL BASIC FOR APPLICATION – VBA

I remember how happy I was when my first programs with GWBasic and Turbo Pascal went through without any errors.

Extensive functions and analysis capabilities make working and studying easier because you get a lot of data and values from different sources to evaluate.

However, with the functions already integrated in Excel you will reach the limits of the Excels.

Operation in Excel is a one-way street. Once you have entered, you either have to stop or drive to the end of the one-way street. It is very rigid, one process after the other.

With the help of VBA programming, VISUAL BASIC FOR APPLICATIONS, or VBA for short, Excel becomes more powerful and the user is free and flexible.

It can be used to program both small macros and object-oriented applications or tools.

VBA is a very useful and easy-to-learn programming language. Since VBA is also a (foreign) language, constant practice is necessary. As you know, practice makes perfect!

The best way to learn programming is to record simple Excel operations. This will familiarize you with the logic used by VBA syntax.

Download free eBooks at bookboon.com

2 DEVELOPER ENVIRONMENT

The developer environment is a parallel world of Excel. It is a stand-alone program, with its own window system and its own toolbar.

How do I get there?

There are two ways to access the developer tools:

- Using the key combination "Alt+F11"
- Through the menu ""Developer" as show below:

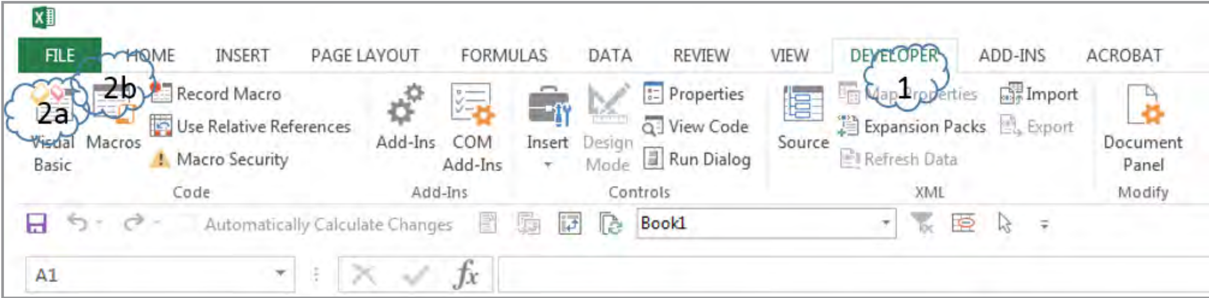


Figure 1: Starting of developer

By clicking on "Visual Basic" (Fig.1, 2a) we start with a new macro. For editing existing macros (Fig. 1, 2b) by clicking on "Macros".

➪ If the "Developer Tools" menu is not visible, select "**File | Excel Options | Developer Tools** ".

The VBA editor is not visible in the development environment the first time you open the Excel file. This should be done via menu "Insert | Module ".

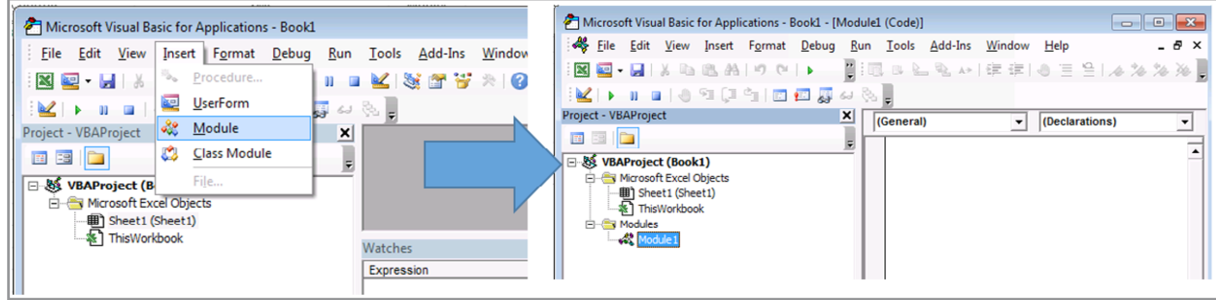


Figure 2: Starting of VBA-Editor

Download free eBooks at bookboon.com

2.1 VBA-EDITOR ENVIRONMENT

The Microsoft Visual Basic window can have different surfaces at the same time which can be faded in and out as desired.

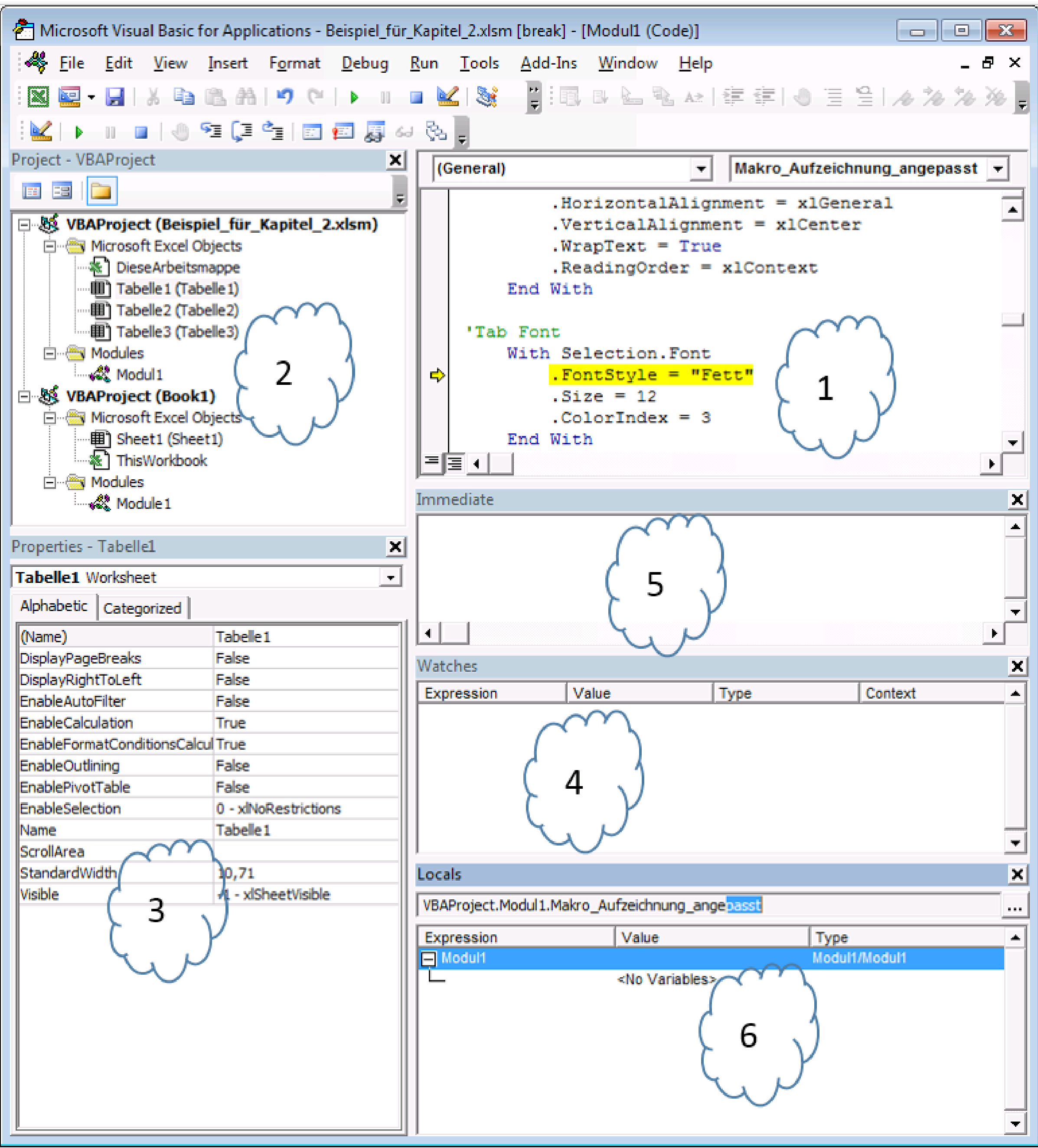


Figure 3: VBA-Surfaces

Names of the window areas:

1. VBA-Editor
2. Project Editor
3. Properties
4. Watches
5. Immediate
6. Locals

Now we can look at them in detail.

2.1.1 THE VBA - EDITOR

The most important segment of the VBA environment is the VBA program editor. VBA provides its own windows for each workbook, for each sheet and for the forms. Here the program code of the modules is entered. Recorded or manually created program parts can be called up, adapted or even modified here.

The VBA editor is not visible when you first open an Excel file. We find it in the "**Insert | Module**" menu.

The editor writes program sections with individual commands, definitions, and comments according to VBA syntax rules. If the syntax of the VBA code is correct, the first letters are converted to uppercase. Otherwise there is a typo.

The operation in this window is similar to a word processor. However, it has a few peculiarities which are listed below:

- Automatic insertion of blank lines,
- Color representation of fonts (Comment | Command | Function | Methods),
- Line break with "space" possible.

Download free eBooks at bookboon.com

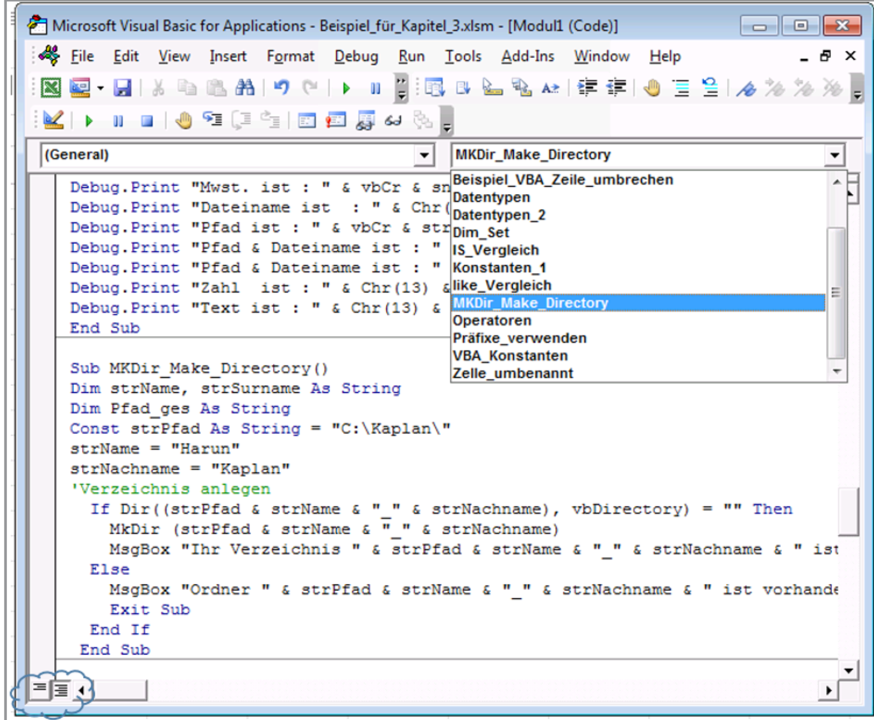


Figure 4: Change between Procedure View & Full Module View

Figure 4 (bottom left) shows two selection boxes. Depending on which function is selected, the editor content is shown either one below the other or individually!

In the menu **Extras | Options**" font color, font size, text input and many other properties can be set as desired in the VBA editor.

2.1.2 PROJECT-EXPLORER

The presentation of the Project Explorer may be familiar from previous experience with Windows Explorer. All open VBA projects, such as Excel files with all tables, modules, diagrams, and Pivot are visible here. These projects can be opened and closed with a plus or minus sign.

As shown in Figure 5, after opening the Project Explorer, you will see "**ThisWorkbook**" and at least one "Table".

When the development environment starts, it opens by default. However, if it should be missing, you can use the menu "**View | Project Explorer** "or use the keyboard shortcut **"Ctrl + R**".

Download free eBooks at bookboon.com

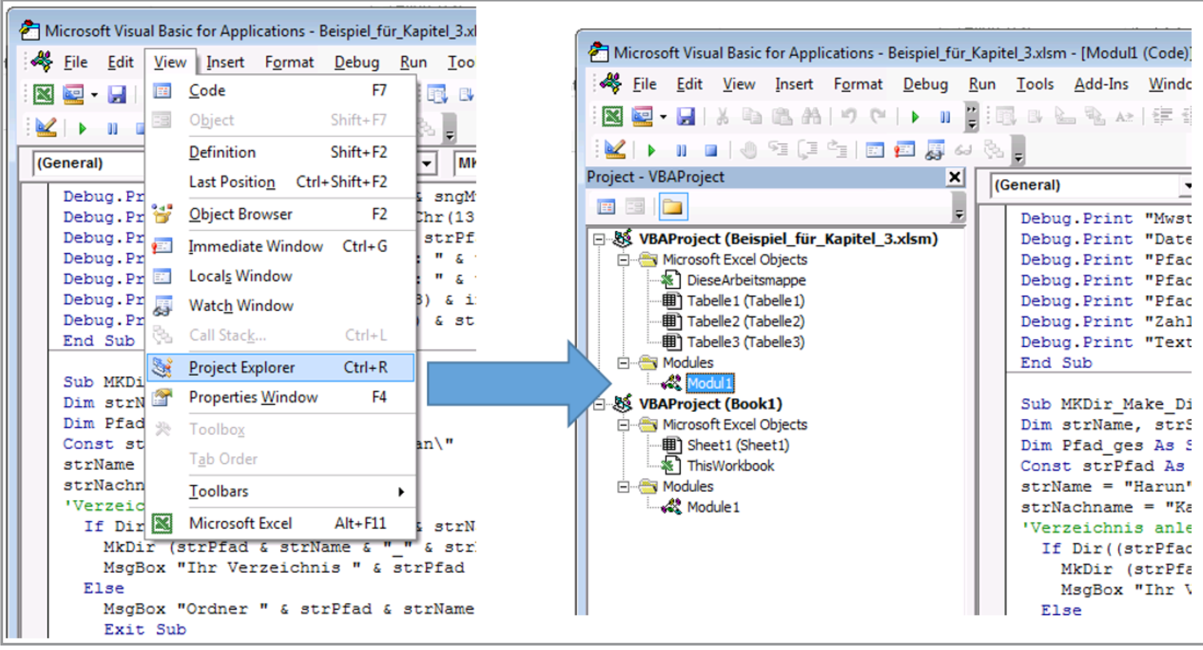


Figure 5: Project

2.1.3 PROPERTIES WINDOW

The Properties Window displays all of the properties or attributes of an object, such as a table, a UserForm, or a diagram. These can be customized as needed.

Callable via

- Menu "**View | Properties Window** "
- Function key "**F4**"

These attributes can be changed either directly or in the VBA procedure. This will be illustrated later in various examples.

The following figure shows two different design representations of a combo box properties window.

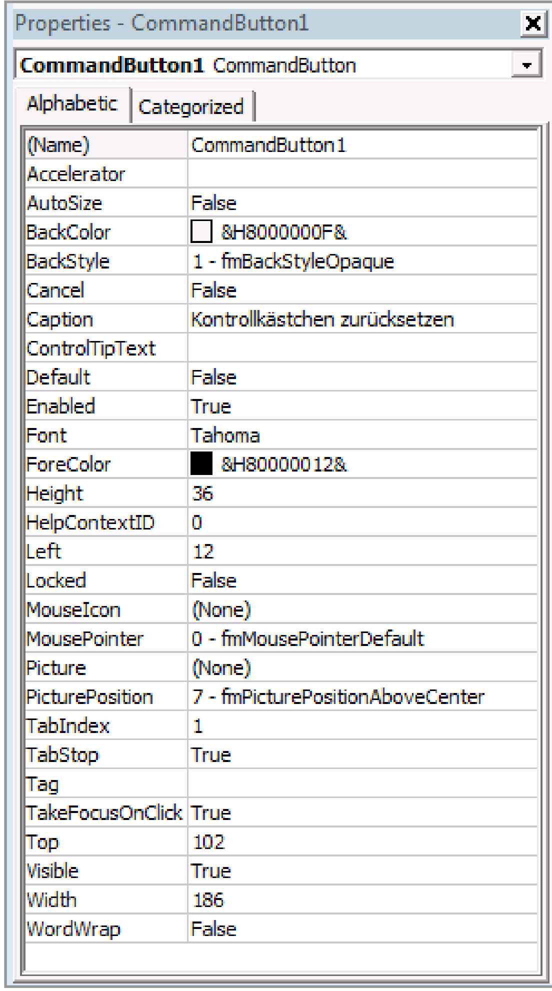


Figure 7: Without group

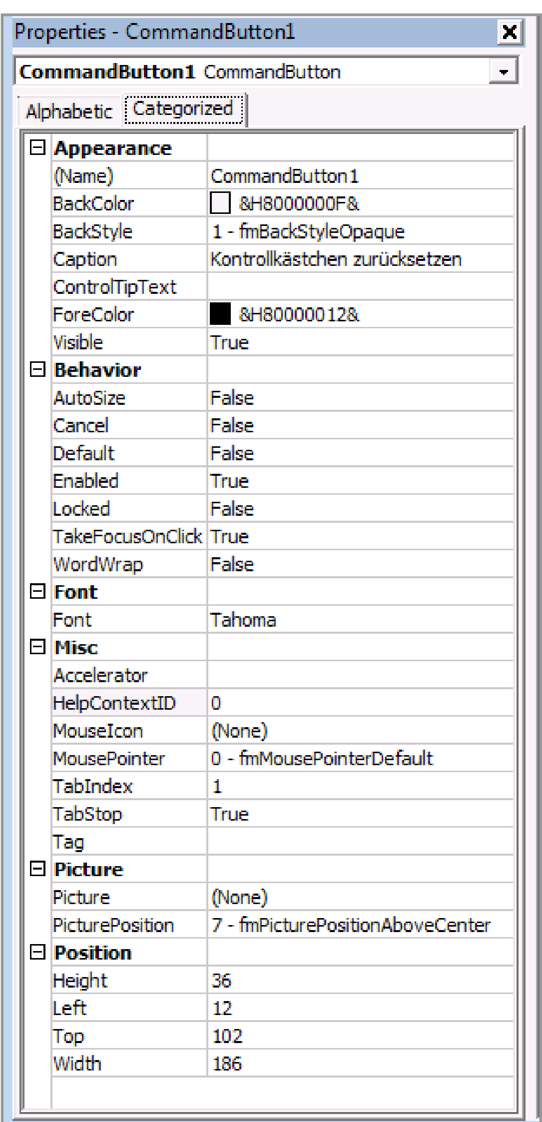


Figure 6: With group

2.1.5 WATCH WINDOW

This window is very useful if testing a macro. There are added terms to monitor here. When the macro expires, we see what value this expression has.

The monitoring window is called up via the VBA menu command "**View | Watch Window** ".

Download free eBooks at bookboon.com

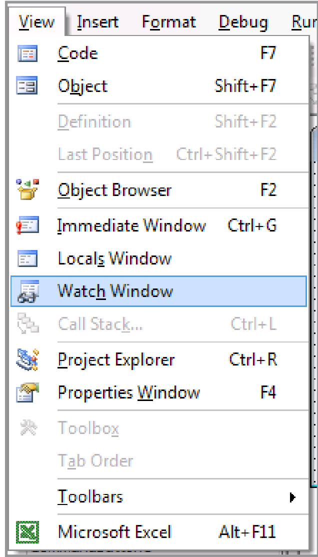


Figure 8: Starting of watch window

Next we add an expression to the Watch Window:
Via menu "**Debug | Add watch** ". Once a variable in the "**Add Watch**" is confirmed, the watch window appears:

Click on the ad to read more

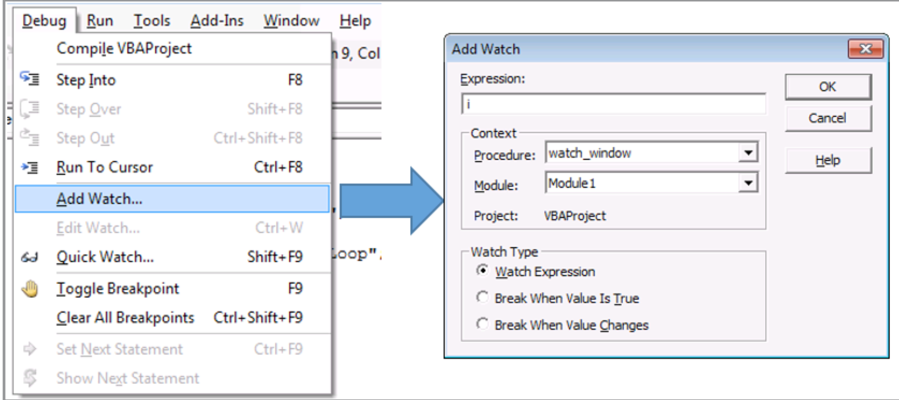


Figure 9: Insert an expression in add watch

Or we mark the item to be monitored and drag it with the mouse into the Watch Window, which is shown in Figure 10.

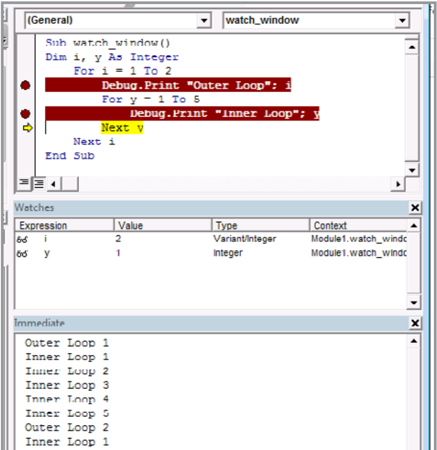


Figure 10: Result in add Watch Window

2.1.6 IMMEDIATE

The Immediate Window has the following functions:

- Execute directly entered VBA statements
- Display determined values of the variables

The variables start in the VBA code with the **Debug.Print**".

You can open this window via menu "**View | Immediate window**" or with the key combination "**Ctrl + g**".

Download free eBooks at bookboon.com

```
Sub Immediate_window()
 Dim intValue, a, b As Integer
 Dim strText As String
  a = 2
  b = 5
  intValue = a + b
 strText = "Microsoft Visual Basic - Excel"
  Debug.Print "The Sum of values " & a & " und " & b; " are = " & intValue
  Debug.Print "My Text is: " & strText
End Sub
```

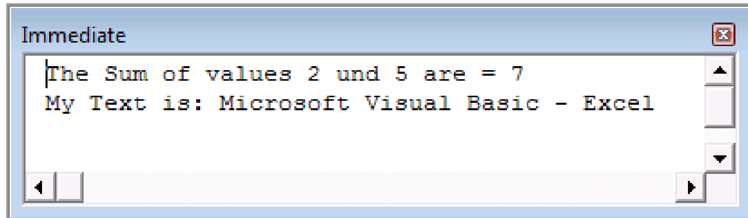


Figure 11: Result in Immediate window

2.1.7 LOCALS WINDOW

In this window, all variables in the VBA code are monitored simultaneously with the determined values. When the VBA code is started with F8 "Step to Step", all defined variables are displayed in the local window with their current values and in defined dimensions.

The local window is opened via menu **View | Locals window** ".

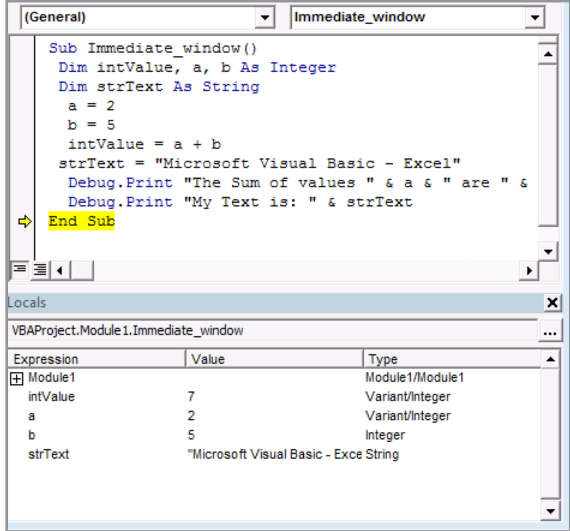


Figure 12: Local window with elements to be monitored

3 "LEARNING BY RECORDING" RECORD OF MACRO

The best way to learn VBA programming in Excel is through macro recording.

All Office programs have a macro recorder which allows all actions to be recorded. The resulting macros are the preliminary stage of VBA programming.

Therefore, it is useful to initially record and optimize many macros in the beginning. That is why I call this experience "learning by recording". The more macros we record and then optimize, the faster we will get into the world of VBA programming and feel "at ease" there.

The recorded macros are used to automate the Excel operation. Recording records all consecutive actions. The recording is rigid and runs exactly as it was also recorded.

The recorded macros do not allow dynamic actions.

A recorded macro is indeed a very useful tool, unfortunately only with limited automation of processes. Therefore, we quickly reach the limits of macro recording.

The key combination **ALT + F11** leads us directly into the area of the editor. In the recorded macro, the syntax and logic of Excel can be analyzed. Change one or more values and you can see the differences after restarting

There are two types of records:

- **Absolute recording**
  A rigid recording, regardless of the current position of the cursor the same cells are always addressed.

- **Relative recording**
  Likewise a rigid recording, but the recorded macro will be executed from the cursor position.

3.1 STRUCTURE OF A PROCEDURE

A procedure is nothing more than a macro that starts with Sub and ends with End Sub. The name of a procedure must not exceed 255 characters and ends with empty parentheses.

Download free eBooks at bookboon.com

The non-empty parentheses, brackets with arguments, will be discussed later in the event procedures.

The procedure mainly consists of two parts, which are listed below:

- Procedure header or also the declaration part: Here the variables occurring in the procedure are declared.
- VBA code or program code: This is where all the music plays.

Depending on the setting, the term "**Option Explicit**" is visible. This forces us to declare all variables that occur in procedures. We need to know in advance what variable a text is and what a number is.

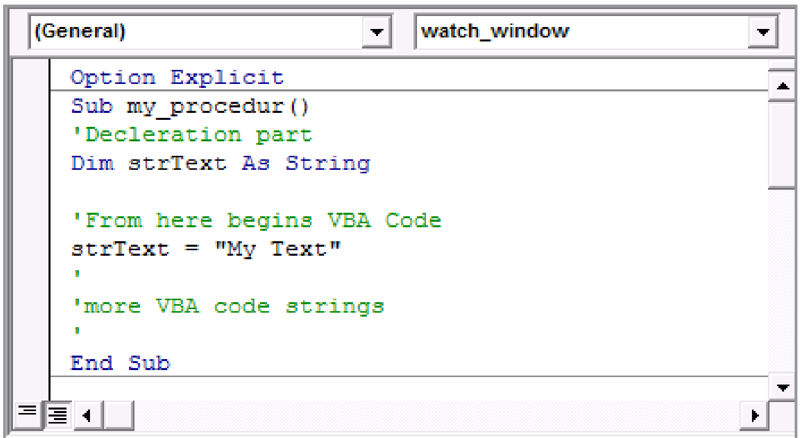


Figure 13: Structure of a procedure

Now we can record our first macro.

3.1.1 THE RECORDING OF MACRO

The macro recording is made via the menu command "**Developer | Record Macro** ". Alternatively, the switch on the VISUAL BASIC toolbar can be used.

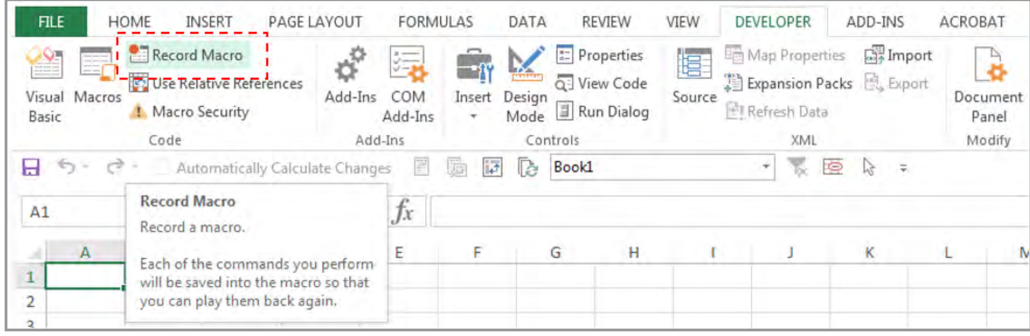


Figure 14: Starting of recording macro

Download free eBooks at bookboon.com

When the recording is started, the **Record Macro** dialog box appears. If you wish, we can fill in the following information about the macro before starting:

- **Enter the macro name**
    - Max 255 characters long
    - Name must begin with a letter
    - "Umlaut" (ä,ö,ü) in the name is permitted
    - Spaces and Hyphens are not allowed
- **Define key combination for starting the macro (start alternative)**
    - By entering the letter "**a**" in the example above, the **shift-key** was held down.
- **Enter the macro description**
    - Information about the function of the macro
- **Select the macro mapping in the "Store macro in" drop-down-list.**
    - If a macro is to be assigned to the work in which it was recorded. "This workbook" is selected here. The personal macro workbook selection makes the macro widely available in all workbooks.

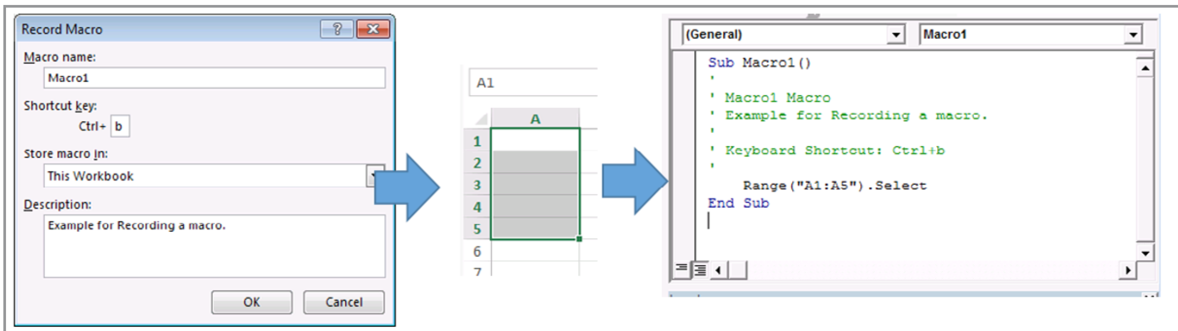


Figure 15: Recording Macro and Listing.

After pressing the OK button, the recording is started. I have marked cells A1 through A5. The list is shown in Figure 15.

The recording of a procedure is similar to a toddler who explores and gets to know its surroundings by touching. Again, you learn the individual commands and their spelling.

3.1.2 RECORDING EXAMPLE

We start our macro recorder via the menu "**Developer | Record macro** ".

Download free eBooks at bookboon.com

First we mark the area "A1: C4", then we call the mask "Format cells". There we define the settings in the "Numbers" tab to three decimal places, in the "Font" tab to "Bold" and a font size of "16".

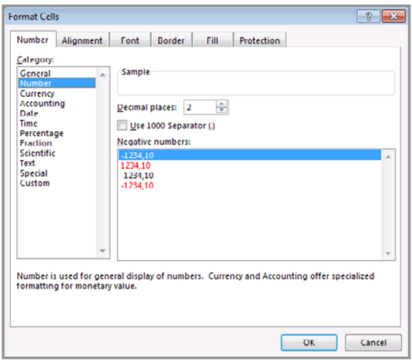


Figure 16: "Format Cells"- window

Our record looks like this:

```
Sub Macro_reckording()
'Macro1 Macro
'Example for Reckording a macro.
'Keyboard Shortcut: Ctrg+b
Range("A1:C4").Select
    Selection.NumberFormat = "0.000"
    With Selection.Font
        .Name = "Calibri"
        .FontStyle = "Fett"
        .Size = 16
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = xlUnderlineStyleNone
        .ThemeColor = xlThemeColorLight1
        .TintAndShade = 0
        .ThemeFont = xlThemeFontMinor
    End With
End Sub
```

Unmodified formatting will either have the default settings or they will have the following entries, such as "**False**"; "**XlNone**"; "**XlAutomatic**"; "**0**"; or "**None**" ending. These unnecessary additional entries make our macros longer and more confusing.

Now we adapt our record regarding clarity:

```
Sub Macro_recording()
Range("A1:C4").Select
   Selection.NumberFormat = "0.000"
'Tap font
   With Selection.Font
      .FontStyle = "Fett"
      .Size = 16
   End With
End Sub
```

3.1.3 CHANGE AN ABSOLUTE RECORDING TO A RELATIVE RECORDING

An absolute recording rigidly formats certain areas. A relative recording is started from the current cell.

We can use the previous absolute record as an example. We will now convert it into a relative record. For this purpose, the statement **ActiveCell** is entered in front of the range ("D10: F12") Select the statement and rewrite the range to "A1: C3".

The ActiveCell statement places the zero point in the active cell. The first column on the right is theoretically column A, then column B and so on. The same rule applies to the lines.

Absolute Recording

```
Sub Makro_Recording()
   Range("D10:F12").Select
.
.
.
End Sub

Now, hire is the Relative Recording:
Sub Makro_Recording()
   ActiveCell.Range("A1:C3").Select
.
.
.
End Sub
```

I would like to repeat my adage: the more we record and analyze the recording, the more we master VBA programming.

Download free eBooks at bookboon.com

3.1.4 ABSOLUTE OR RELATIVE RECORDING

There are two types of records: an absolute and a relative record. As described in the previous chapter, absolute recordings always address the same cells.

The relative recording is recorded like the absolute recording. The only difference is that the button "**Use relative references**" is selected before the recording.

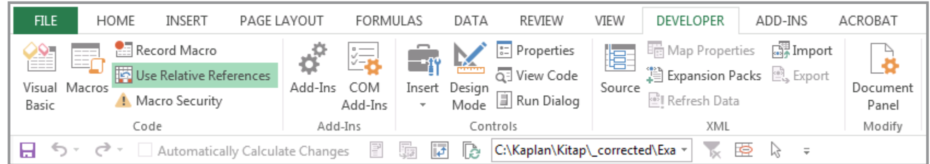


Figure 17: Starting of "Use Relative References"

Now a macro with the same content and with different recordings.

- The area D10: F12 has been marked
- Defined with a yellow background color

In the absolute recording, regardless of the cursor position, the areas D10: F12 are always marked and formatting is performed.

```
Sub Absolute_recording()
' Absolute_recording macro
   Range("D10:F12").Select
   With Selection.Interior
      .Pattern = xlSolid
      .Color = 3
   End With
   Range("D10").Select
End Sub
```

In the relative record, starting at the cursor position, the next three column cells and the next three row cells are marked and formatting is performed.

```
Sub Relative_recording()
' Relative_recording macro => show Figure 18
   ActiveCell.Range("A1:C3").Select
   With Selection.Interior
      .Pattern = xlSolid
      .Color = 3
   End With
   ActiveCell.Select
End Sub
```

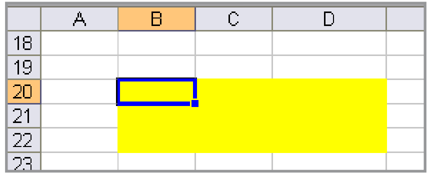


Figure 18: Relative recording

3.2 DELETE, EXPORT OR IMPORT OF MACROS / PROCEDURE

A recorded macro is opened via the menu command "**Developer | Macros** ". Click on the "**Delete**" button in the dialog box to delete a selected macro.

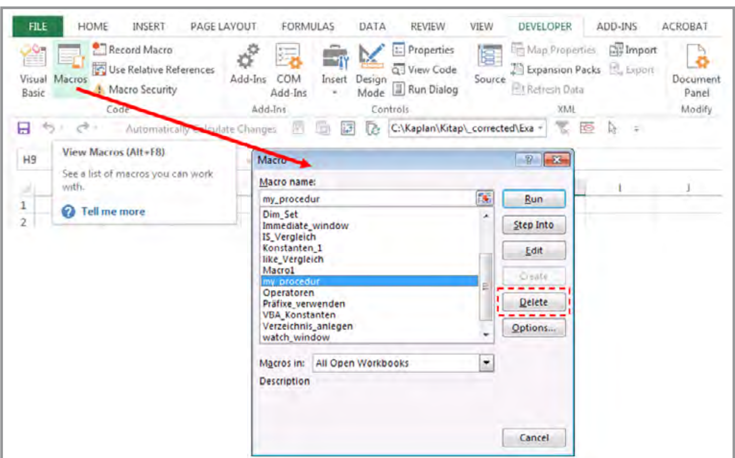


Figure 19: Delete of a macro

3.2.1 EXPORT (REMOVE) AND IMPORT OF MODULE

In order to remove an entire module in the VBA, this must be selected and then removed either via Variant_1 or Variant_2, from figure 20. The query "**Do you want to export Modul1 before removing it?**" Has to confirm with "**No**".

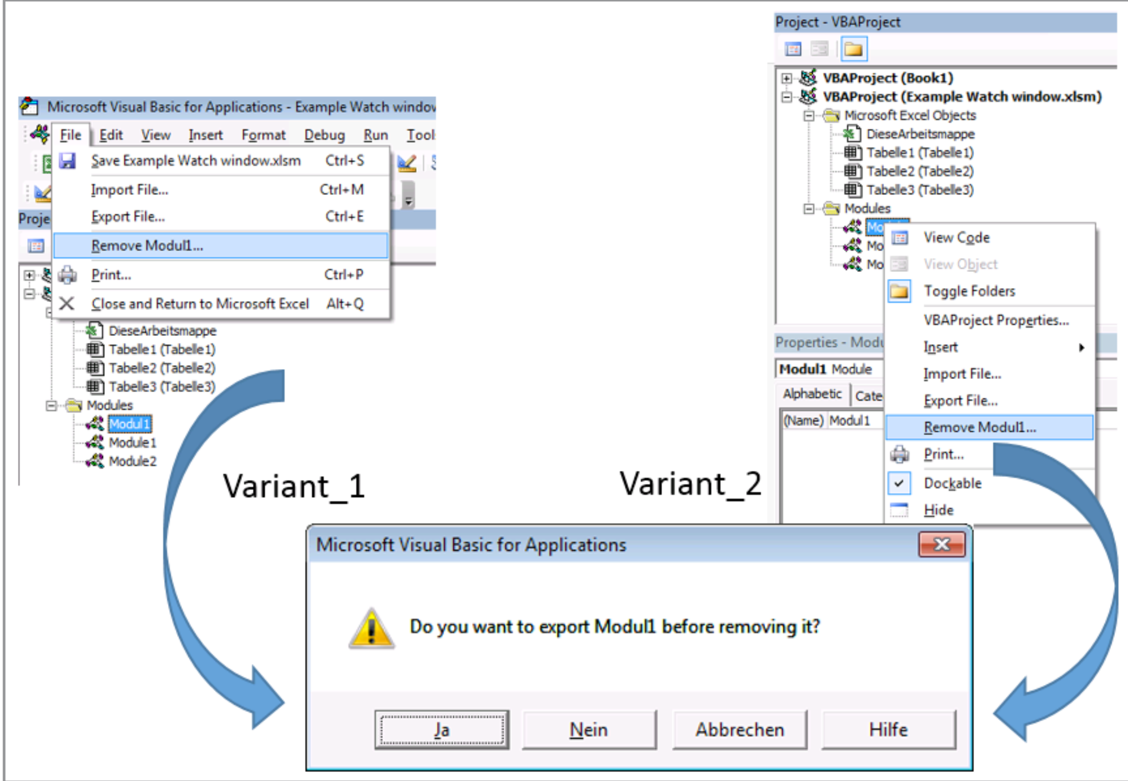


Figure 20: Exporting or Deleting of Macros

For export, we proceed as if for deletion. Only now, the delete query will be answered with "**Yes**". The exported macro is saved as a basic file * .bas.

These or similar basic files can be imported again at any time. To do this we select the menu command **File | Import File**" or the shortcut "**Ctrl + M**".

3.3 START A MACRO

Macros can be opened in several different ways:

- From the editor
- By keyboard shortcut
- By assignment to a key or to an object
- After a certain time
- Function keys
- Through the call statement in procedure and
- Start with your own menu programming.

Download free eBooks at bookboon.com

3.3.1 STARTING BY EDITOR

In VBA editor, position the cursor somewhere in procedure, then

- by pressing the F5 key or by pressing F8 for a step by step sequence
- Click the blue arrow in the toolbar or,
- Via the menu command „**Execute | Sub | User Form**“.

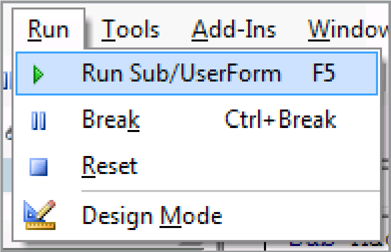


Figure 21: Starting from Editor

3.3.2 SHORTCUT: CTRL + SHIFT + {CHARACTER}

If you have not already done so, we can define a keyboard shortcut when starting a macro. Later, this macro can also be opened with this combination. This assignment can be written with uppercase or lowercase letters.

Subsequent assignment of a key combination looks like this:

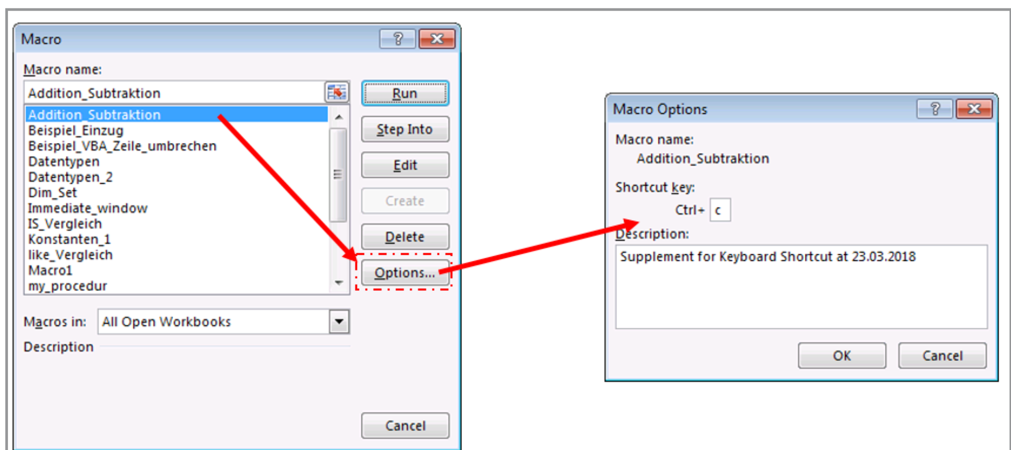


Figure 22: Start a macro by shortcut

Download free eBooks at bookboon.com

3.3.3 ASSIGN MACRO TO A BUTTON

Such buttons can be created from a toolbar of the form, the Autoforms, or the controls Toolbox.

To assign a macro, proceed as follows:

- Mark the button
- Right-Click on the "Assign Macro" window
- Select macro
- Confirm with OK-button

Discover the truth at www.deloitte.ca/careers

3.3.4 ASSIGN MACRO TO AN OBJECT

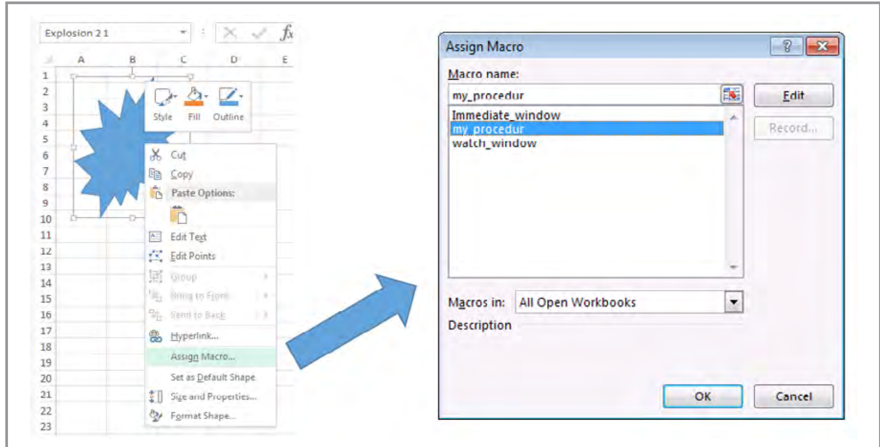


Figure 23: Assign macro to an object

3.3.5 STARTING A MACRO IN WORKBOOK_OPEN()-EVENT

"**Workbook_open** ()" event is stored in the editor of "**ThisWorkbook**". Here I will only give a short explanation with figure 24. We will see futher examples later.

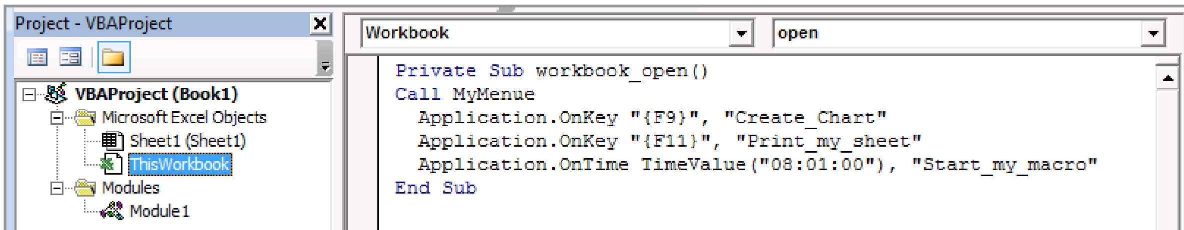


Figure 24: Instruction in workbook_open()-Event

- Syntax for starting at **a specific time**:
  Applicatio.OnTime Timevalue ("Hour: Minute: Second"), "*Macro to invoke*"

- Syntax for start **via function key**:
  Applicatio.OnKey "{Function key},"*recorded macro*"

- Syntax for the start **via call statement**:
  Call <*procedure name*>

Download free eBooks at bookboon.com

3.3.6 STARTING A MACRO FROM AN EXTERNAL EXCEL FILE

The "**Application.Run**" statement will open a macro or function from an external Excel file.

In our example, "MacroName" is executed from the Excel file "Filename".

Application.Run "'" & Filename & "'!MacroName"

3.3.7 THE SECURITY

The security settings are an important topic in Excel. Any open Excel file may contain dangerous macros. Therefore, it is recommended to set the security setting to at least the middle position. When set, it will open a dialog box when opening an Excel file that contains macros. In it we can still decide whether to open the file call with or without macro activation.

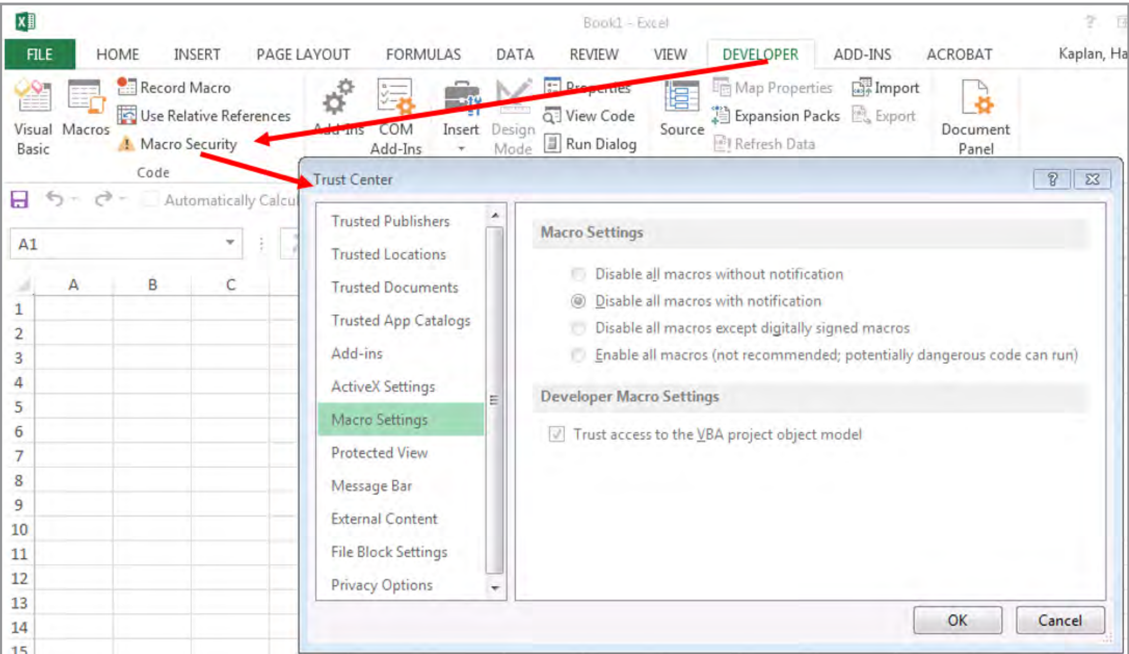


Figure 25: Security settings

You get these when the menu command "**Developer | Macro Security** "is displayed. The Security dialog allows you to create different settings.

4 SCRIPT CONCEPT BY VBA

Before delving into the world of VBA language elements, I would like to explain some prerequisites for VBA programming.

VBA Visual Basic for Applications is an object-oriented macro programming language with a very extensive set of functions and instructions for creating stand-alone programs.

With the VBA programming we can extensively exhaust the possibilities of Excel. Reasons for this can be the following:

- Automation of standard tasks.
- Function extension by "own" functions.
- Programming a user interface (UserForm) as Excel application tools.

As you know, the way to love is through the stomach. This analogy includes vegetables, meat, fish, a few different spices, and so on to other ingredients. All these ingredients are mixed according to our wishes in a specific order to make a hopefully delicious meal out of it.

The path to VBA programming is through macro recording. During macro recording, all activities of the user are recorded.

With VBA, we can address from the top level down to the lowest level. These levels are:

- Excel application as an **Application**
- Workbooks as a **Workbook**
- Excel sheet as a **Sheet**
- Individual areas as a **Range**
- Cells itself as **Cells** / **Range**
- Properties of cells, for example font color as **FontColor**.

4.1 PROCEDURE, MODULE, VBA-CODE

We have already seen the chapter contents in chapter 1. Here we will see it again briefly, as the following topics are built on it. Procedures are groups of instructions and have a solid framework. I refer to it as the "head and foot" of the VBA program.

Download free eBooks at bookboon.com

They start with the statement Sub Name_Procedure () and end with End Sub:

```
Sub Say_hello_1()
        'You can write VBA command here
End Sub
```

4.1.1 SYNTAX OF VBA-CODE

An instruction is a syntactic command for definitions, declarations or operations. These instructions can be grouped by shifting, making them easier to check. By indenting, the listing can be presented in a clearer, more structured way. The more we click on indent, the more indentation. The reverse function is with outdent. This allows, for example, With … End With; For … Next; If … End If; Select … End Select, to become much clearer.

You can do this with icons in the Edit toolbar in the VBA Editor. With the option "**Enlarge indentation**" the current line or several marked lines are moved to the right, with "**Reduce indentation**" the lines are movied to the left.

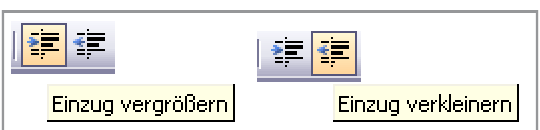


Figure 26: Indenting (Tab) / Outdenting (Shift+Tab)

An example with outdenting:

```
'Outdenting
For x = 1 To 5
If name = "Harun" Then
Range("A" & x).Value = name & "_" & x
Else
Range("B" & x).Value = name & "_" & x
End If
Next x
```

An example with indenting:

```
'Indenting
For x = 1 To 5
  If name = "Harun" Then
    Range("A" & x).Value = name & "_" & x
  Else
    Range("B" & x).Value = name & "_" & x
  End If
Next x
```

4.1.2 UPPER- AND LOWER-CASE

The VBA code for names of subs, functions or variables can be written case insensitive. As soon as a line is finished, VBA converts the first letters of the commands, functions, arguments into uppercase letters.

4.1.3 ARGUMENTS IN FUNCTIONS UND METHODS

Methods are enumerations in parentheses of a function. They are separated in Excel with the semicolon. In VBA syntax, however, they are separated by a comma.

For this an example function “COUNTIF ()”: Between the brackets, you first enter the range and then the searched term. They are separated by “;”. In VBA, however, with “,” separated.

The term “Stuttgart” should be counted in column C of an Excel spreadsheet. The result should appear in cell “D1”.

In Excel:

- =CountIf(C:C**;**”Stuttgart”)

In VBA:

- Range(“D1”).Value = WorksheetFunction.CountIf(Range(“C:C”)**,** “Stuttgart”)

4.1.4 COMMENTARY LINE

Sections of text starting with an apostrophe are marked as comments by the VBA editor. They are not executed.

You can add your comments as a line or at the end of a statement.

Lines can be summarized either individually or as a comment area. Simply mark the area and select the option “Comment out block”.

Download free eBooks at bookboon.com

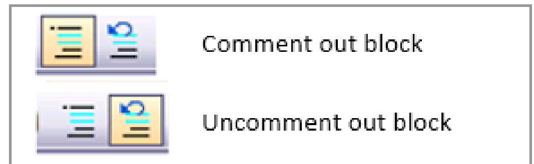


Figure 27: Annotate a VBA-Code

```
Sub Example_Annotate_with_Apostrophe()
'With Indenting
'Stand: 16.06.11, Ka
For x = 1 To 5 'This loop run 5x
  If name = "Harun" Then 'Control to
    Range("A" & x).Value = name & "_" & x
  Else
    Range("B" & x).Value = name & "_" & x
  End If
Next x
End Sub
```

4.1.5 LIST PROPERTIES AND METHOD – INTELLISENSE

The editor writes program sections with individual commands, definitions and comments according to VBA syntax rules. The first few letters of a command immediately display the IntelliSense collection as a pop-up appear. Even if you have entered an object with a period, this IntelliSense list or a selection window will display the possible commands, methods, or properties for that object. That is a relief in programming.

IntelliSense is a VBA-supplied tool for automatically completing commands in source code.

If it does not work, "**Ctrl + j**" or "**Ctrl + Spacebar**" can help.

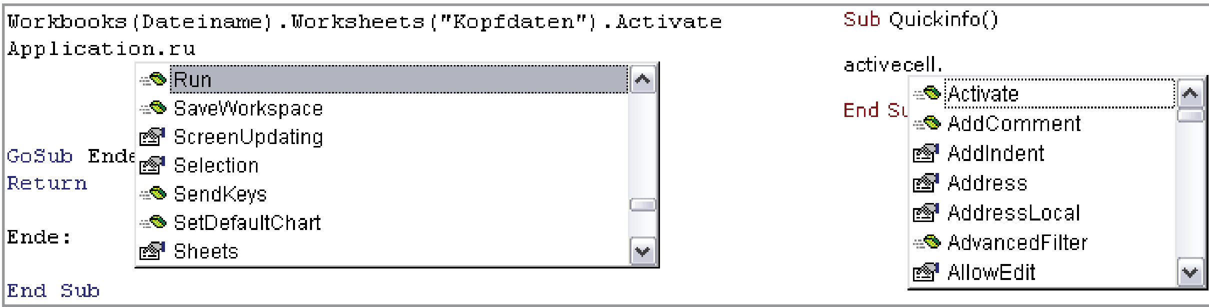


Figure 28: Example IntelliSense

The IntelliSense collection also appears after an equals sign of an object, but constants are offered this time.

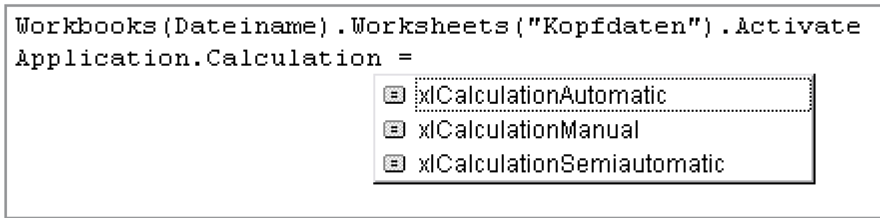


Figure 29: IntelliSense by a constant

4.1.6 QUICK INFO

Quick help for the syntax of functions, methods or procedures is a good help. If, for example, a method is entered, a short reference with the necessary and possible arguments appears directly below the insertion point after entering the next empty space or an open parenthesis.

The arguments in square brackets are optional. All others are to be entered.

- Example 1:

```
Sub Quickinfo()
activecell.offset(
End Sub    Offset([RowOffset], [ColumnOffset]) As Range
```

Figure 30: Quick info-Example of „Offset"

The tooltip of "Offset" tells us that after an open parenthesis all necessary arguments are listed. Commonly needed arguments are in parentheses and the currently needed argument is shown in bold type.

- Example 2:

```
Sub Quickinfo()
ActiveCell.Offset(0, 0).Select
msgbox(
MsgBox(Prompt, [Buttons As VbMsgBoxStyle = vbOKOnly], [Title], [HelpFile], [Context]) As VbMsgBoxResult
End Sub
```

Figure 31: Quick info-Examples of MsgBox with arguments

In our second example, we used the MsgBox function.

If you run your procedure step-by-step with F8, you will also get tool tips for the current VBA line. This is marked in color by positioning the mouse pointer.

In the following example, the variable "Response" is assigned the address of the active cell. When we move the mouse pointer to "answer", the assigned value appears as a tooltip.

```
Sub Quickinfo()
Dim Antwort As String
Antwort = ActiveCell.Address
MsgBox (Antwort)
    Antwort = "$B$1264"

End Sub
```

Figure 32: Show a variable as Quick info

4.1.7 GAP OR CONCATENATE OF A MULTI-LINE IN VBA-SYNTAX

Long or very long VBA instruction lines can't always be avoided in the VBA editor. A space combined with an underscore helps us to present our programming in a more structured way. The underscore is entered immediately after the space. This tells the VBA editor that the statement line has not yet ended.

- **A line break is realized with a combination of a space and an underscore!**

This has the advantage that the entire VBA line is visible. However, the break can't happen anywhere. It is only allowed in front of or behind an element of expression. No comment may be inserted after the underscore. Sequences in strings must be terminated with a quotation mark and linked with the join operator "**&**" and is called concatenation. This can combine two or more strings to from a new string.

The following example illustrates both variants.

Download free eBooks at bookboon.com

```
Sub Example_VBA_Syntax_gap()
Dim strName, strfirstname, strAddress As String
strname="Max"
strfirstname="Mustermann"
straddress="Musterstrasse 12" & chr(10) & "70000 Stuttgart"

'Without gap
MsgBox ("Name: " & strname & Chr(10) & "First Name: " & strfirstname & Chr(10) & "Address: " &
straddress)

'With gap
MsgBox ("Name: " & strname & Chr(10) & _
          "First Name: " & strfirstname & Chr(10) & _
           "Adresse: " & straddress)
End Sub
```

4.2 VARIABLE, CONSTANT AND DATE TYPE

Variables are placeholders for numbers, texts, objects and can be changed in the macro process. Variables are used in each programming language so that your values can be recorded during program execution and, if necessary, used later or taken over by another (sub) procedure.

The properties of the variables are set at the beginning of the procedure with the **Dim** keyword. These definitions can be numerical values, texts or objects.

The naming of the variables follows the following rules:

- It starts with a letter
- It can be up to 255 characters long
- It contains no dots, minus signs, spaces or special characters

The variable abbreviations are set by us, but they also indicate their origin. For example, an integer number with "intNumber" means that the variable number belongs to the type integer. Let's take a look at a procedure that uses them:

```
Sub Use_prefixe()
Dim intNumber As Integer
Dim strName As String
Dim blnAntwort As Boolean
Dim bytValue As Byte

  intNumber = 231
  strName = "Harun"
  blnAntwort = True
  bytValue = 1
End Sub
```

This is very helpful in debugging, since variable types declared by these abbreviations can be detected faster. Depending on the declaration type and the declaration location, these variables have a scope that we will look at before presenting the individual variable types.

4.2.1 DATE-TYPE

Data types are groupings of data. These are all kinds of numbers, texts, etc. They are declared at the beginning of the procedure. They are not always necessary for smaller procedures, but are often used in longer procedures or projects with multiple programming.

Here is an explanation of what a data type is:
We assume that the term "people" should be our data. People are all "men, women and children". If we dimension them individually, it would look like the example shown below. All "men, women and children" belong to the species "human".

- **Dim** men, woman, child **As** human

In other words, there are many worksheets, that is, worksheets (plural). These belong to the species worksheet, so worksheet (singular).

You can declare it named arbitrarily. In the example "Number1" and "Number2" are declared as Integer, and "Number3" and "Number4" are declared as Long. When used in a procedure, they would not be unique, which are integer type and which are long type.

It is better if you can immediately recognize what type of data it is. Therefore, before each variable we have an abbreviation, prefix with:

Dim ***int****Number1*, ***int****Number2* **As Integer**
Dim ***int****Number3*, ***int****Number4* **As Long**

VBA knows the following data types (variable types):

Data type Prefix	Descripting
Array arr	An array is a data structure that contains several variables of the same type.
Byte byt	Holds unsigned 8-bit integers that range in value from 0 through 255.

Data type Prefix	Descripting
Boolen bln	Holds values that can be only True or False.
Currency cur	Fixed-point number with 15 digits in front and 4 digits after the decimal rounded.
Date dat	Date and time.
Double dbl	Floating point number with 16 digits precision, positive and negative values.
Integer int	Numbers with values ± 2.147.483.647.
Long lng	Numbers with values ± 9.223.372.036.854.775.808.
Single sng	Floating point number with 8 digits precision, positive and negative values.
String str	Holds sequences of unsigned 16-bit code points that range in value from 0 through 65535.
Object obj	Holds addresses that refer to objects.
Variant var	Numbers values or strings.

Table 1: Abbreviation of Date-type with descripting

Download free eBooks at bookboon.com

```
Sub Date_type()
    'With Dim declared
    Dim curValuel As Currency
    Dim blnValue_3 As Boolean
    Dim datDate, datDate_1 As Date
    'With Static declared
    Static strText_1 As String * 5
    Static intValue_1 As Integer
'
    curValue = 250 / 3
    intValue_1 = 250 / 3
    Value_2 = 250 / 3
    '
    blnValue_3 = 250 / 3 < 5 * 5
    '
    strText = "ABCDEFsdasd"
    strText_1 = "ABCDEFsdasd"
    '
    datDate = #5/25/1964#
    datDate_1 = Date

    Debug.Print "The variable has the value as Currency " & curValue
    Debug.Print "The variable has the value as Integer " & intValue_1
    Debug.Print "The variable has the value without declaration " & Value_2
    Debug.Print "The variable has the value as Boolean " & blnValue_3
    Debug.Print "The variable has the value as String " & strText
    Debug.Print "The variable has the value as String with five characters " & strText_1
    Debug.Print "The variable has the value as Date " & datDate
    Debug.Print "The variable has the value as Date " & datDate_1
'
End Sub
```

```
Immediate
The variable has the value as Currency 83,3333333333333
The variable has the value as Integer 83
The variable has the value without declaration 83,3333333333333
The variable has the value as Boolean False
The variable has the value as String ABCDEFsdasd
The variable has the value as String with five characters ABCDE
The variable has the value as Date 25.05.1964
The variable has the value as Date 26.03.2018
```

Figure 33: Example Date-type_1

The next example reads the range A1: C4 and prints in the immediate area.

```
Sub Date_type_2()
    Dim intX, intY As Integer
    Dim objWS As Object
    Set objWS = Worksheets("Sheet1")
    For intX = 1 To 4
      For intY = 1 To 3
        Debug.Print "Array-Range " & objWS.Cells(intX, intY)
      Next intY
    Next intX
End Sub
```

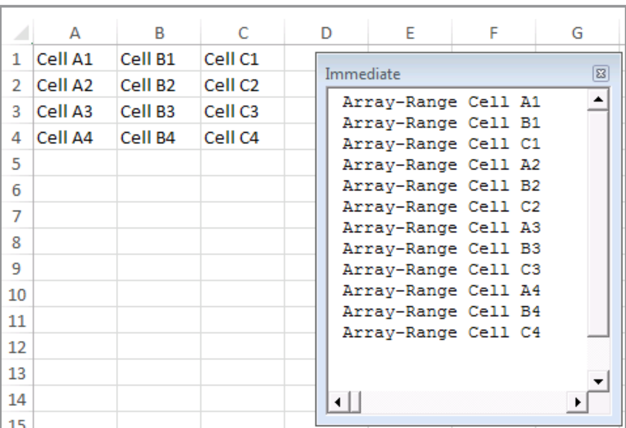


Figure 34: Example Date-type_2

4.2.2 ABBREVIATION OF PREFIX IN VBA-CODE

Prefixes are abbreviations that can be used at the beginning of variable names. There are fixed and variable abbreviations. The built-in constants have fixed abbreviations that indicate their origin. For example:

- vb from VBA-Object-Library
- xl from Excel
- fm from MSForms-Library
- mso from MS-Office
- grd from Toolbox-Sheet.

4.2.3 THE SCOPE OF VARIABLES

A variable can be addressed within the following locations:

- only in one procedure / function (local; procedure-level)
- in a module (module-level)
- in all modules (project-level)

4.2.4 VALIDITY WITHIN A PROCEDURE

In procedures and functions, a simple dim statement is sufficient for declaring a variable. The declared variable is only valid in the current procedure or function. We call it a “local variable”.

```
Sub Validity_Range_1()
Dim intValue As Integer 'Validity within this macro only
intValue = 231
MsgBox intValue
End Sub

Sub Validity_Range_2()
Dim strText As String 'Validity within this macro only
strText = "Validate Range"
MsgBox strText
End Sub
```

Microsoft Excel
231
OK

Figure 35: Validity within a procedure with output

The figure above shows two procedures in “Module1” with an output of the first procedure. We have declared the variable “intValue” in procedure “Gültigkeitsbereich_1” and the variable “strText” in procedure “Validity_Range _2”.

“IntValue” only applies in the upper procedure and “strText” only in the lower procedure.

4.2.5 VALIDITY WITHIN A MODULE

Validity within a module means that a variable is valid in all procedures. This variable is declared outside the procedures.

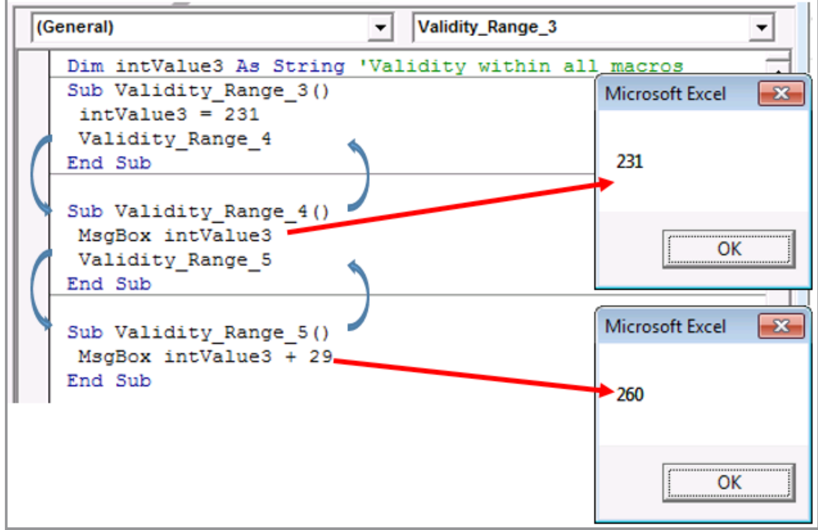


Figure 36: Validity within a module, therefore in all procedures

Download free eBooks at bookboon.com

The instruction "Dim intValue3 As Integer" declares the variable "intValue3" in the module header. Procedures with inter-procedurally valid variables can also be used with the private keyword. The procedure "Validity_Range_4" and "_5" could also be edited with the private keyword:

```
Dim intValue3 As String 'With Dim declared
Private Sub Validity_Range_3()
  intValue3= 231
  Validity_Range_4 Can be written here with or without Call Validity_Range_4
End Sub

Private Sub Validity_Range_4()
  MsgBox intValue3
  Call Validity_Range_5
End Sub

Private Sub Validity_Range_5()
  MsgBox intValue3 + 29
End Sub
```

Click on the ad to read more

4.2.6 PASSING VARIABLES TO A PROCEDURE AND TO A FUNCTION

Passing a variable between procedures or between a procedure and a function is also possible. The variable to be passed is entered with a declaration in parentheses after the designation.

We can look at the first procedure in figure 37. To calculate the circular area from a radius, we pass the radius, here 5 mm, with the line SubProgramm (5) to the procedure "SubProgramm". The calculated value is output formatted in the procedure "SubProgramm".

Here are the examples:

```
(General)                                circular_area
Sub circular_area()
  subprogram (5) 'Radius is 5cm
End Sub

Sub subprogram(dblRadius As Double)
  dblCalculation = (dblRadius * dblRadius) * 3.14 '5x5x3,14
  MsgBox "Circular ares is " & Format(dblCalculation, "0.00") & " cm²"
End Sub
```

Microsoft Excel

Circular ares is 78,50 cm²

OK

Figure 37: Passing variable to a sub-procedure

Now let's look at a function. Here we calculate the circular area by passing the radius with the line "dblResult = Calculation (5)" to the "Function dblCalculation". The calculated value is output in the procedure "dblCalculatation".

```
(General)                                dblCalculation
Sub circular_area_2()
  dblResult = dblCalculation(5) 'Radius is 5cm
  MsgBox "Circular ares is " & Format(dblResult, "0.00") & " cm²"
End Sub
Function dblCalculation(intRadius As Integer)
    dblCalculation = (intRadius * intRadius) * 3.14 '5x5x3,14
End Function
```

Microsoft Excel

Circular ares is 78,50 cm²

OK

Figure 38: Passing variable to a function

4.2.7 VALIDITY WITHIN ALL PROCEDURES OF A WORKBOOK

If the variables are to be valid in all modules, then we declare these variables, for example in module1, with the **Public** statement. That is, these variables are provided by all modules in all procedures.

Our example: We write in module4 “Bsp_Public_Modul1” and declare the variables outside the procedure with the Public statement as “**Public** Value1, Value2, Result **As Integer**”.

We copy this procedure and insert it in module1 and in table1 (table1). We adapt the with the module names. In module1 this is called “Public_module1”, in module2 “Public_module2” and in Sheet1 “Public_Sheet”.

We get the value “32” everywhere as a result.

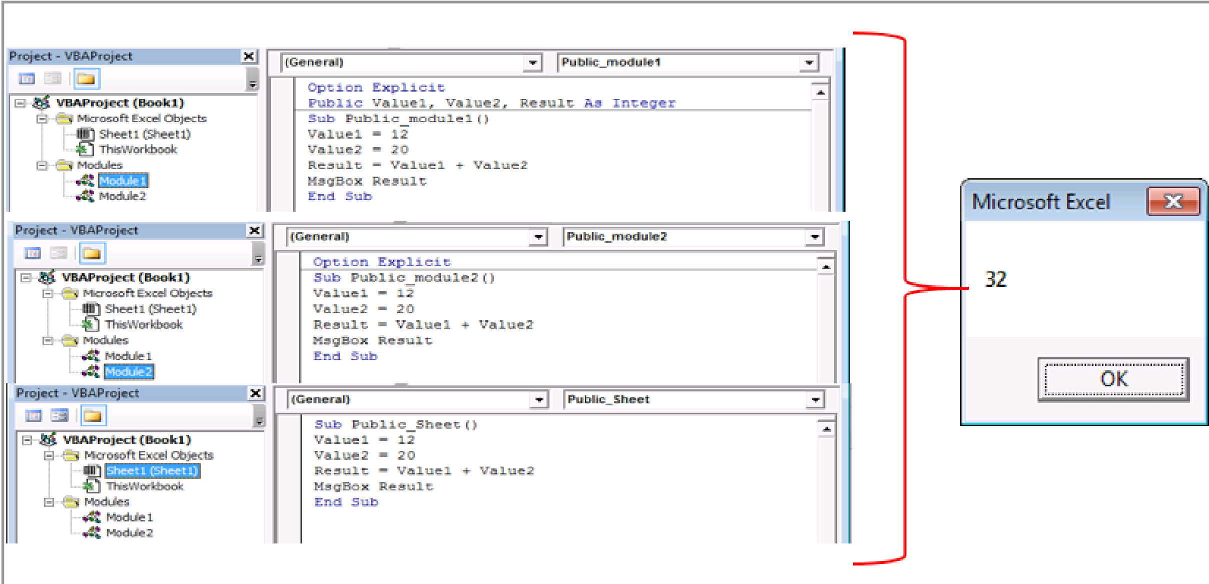


Figure 39: Validity of a variable in workbook

4.2.8 STATIC VARIABLE

Variables with a **Static** statement retain their values even when the procedure is finished. By contrast, variables with a **Dim** statement start from the beginning.

Our example below “Main_Sub” calls the procedures “Normal_Procedure” and “Static_Procedure” five times in succession.

In “normal_procedure”, the x value with the Dim statement is defined as “**Dim** x **As Integer**”. In the “static_procedure” procedure, the y value is defined using the static statement, “**Static** y **As Integer**”.

Let’s go through “MainSub” twice in a row. The x value always stays at “1” and the y value becomes “1” higher.

What is the second run? The x value will still remain “1” and the y value will continue to increase by “1”.

Also, start this “y-value” at “0”

- a runtime error occurs
- the “Static” procedure is aborted
- Excel is finished

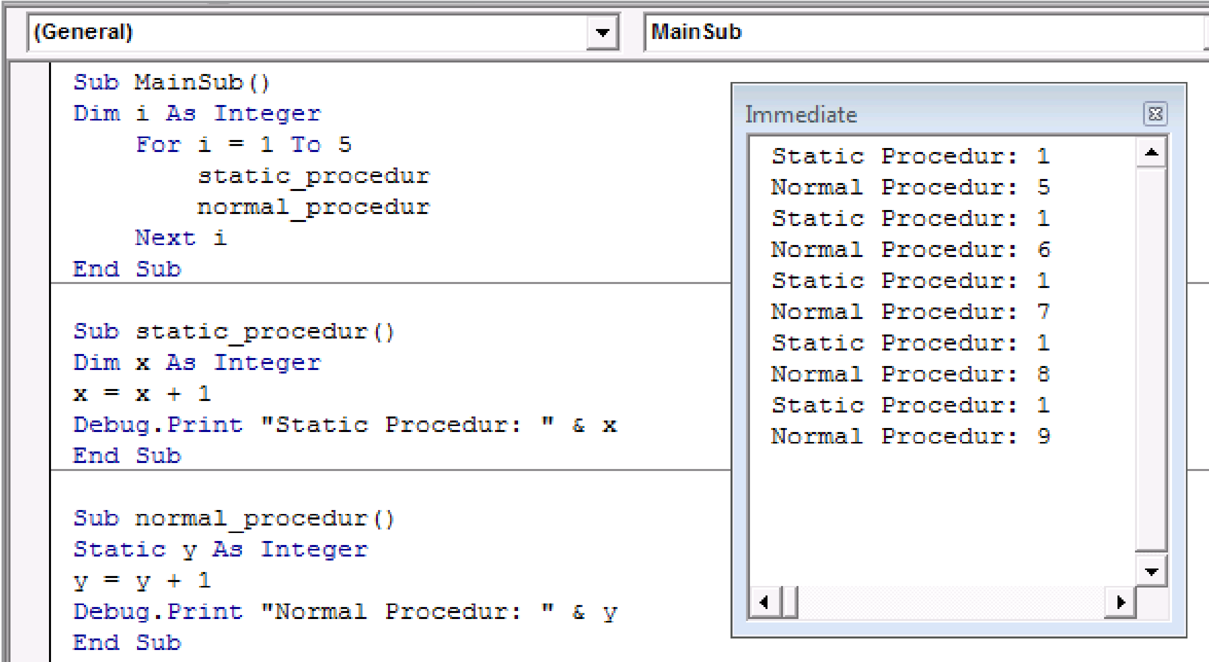


Figure 40: Functionality of a Static-statement

4.2.9 SET VARIABLE OF OBJECT

So far we have declared our variables with the **Dim** statement. Such variables are provided by either an input or a result of calculation or the like.

The **Set**-variables, on the other hand, are fixed components of the excel.

Such Excel objects are:

- Cells / Ranges of a table (Range);
- All diagrams (Charts), tables (Worksheets);
- All folders (Workbooks);
- Forms.

VBA syntax starts with Set, followed by Object variable, then Excel object type. An example:

- Set objWS = Worksheets("Sheet1")

  …

- Set objWS = Nothing

These variables make accessing objects easier and faster.

Here is our example with and without a set variable:

We have two tables, "Sheet1" and "Sheet2". Cell "A1" of Sheet1 has the value "1923". It should be multiplied by the value "2.5" and the product entered in cell "A1" of table2.

```
Sub Exaple_without_Set()
  Worksheets("Sheet1").Range("a1").Value = 1923
  Worksheets("Sheet2").Range("a1").Value = Worksheets("Sheet1").Range("a1").Value * 2.5
End Sub
```

Here is the example with the set statement:

```
Sub Example_with_Set()
Dim Cell1 As Range
Dim Cell2 As Range
'
Set Cell1 = Worksheets("Sheet1").Range("A1")
Set Cell2 = Worksheets("Sheet2").Range("A1")
'
Cell1 = 1923
Cell2 = Cell1 * 2.5
End Sub
```

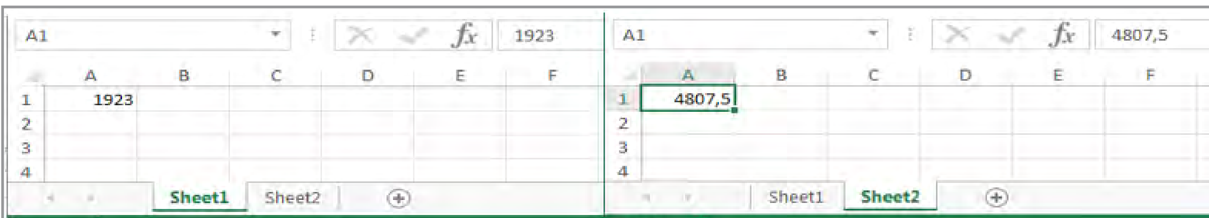


Figure 41: Result Set

4.2.10 USER-DEFINED DATE TYPE

The user-defined data types are assembled from the basic types listed above using the **Type** statement. They are listed between the following keywords:

- ➪ **Private** (valid only where they have been defined) and
- ➪ **Public** (cross-procedures, that means they apply in all procedures).

```
Option Explicit

Private Type Autor
  intNumber As Integer
  strFirstName As String * 15
  strName As String * 20
  strTitle As String * 10
  blnGender As Boolean
  strPostCode As String * 5
  strStreat As String * 20
  strPlace As String * 20
  datBirthday As Date
  strPhone() As String
  strTelefax(1 To 2) As String
  sinSalary As Single
End Type

Private Type Book
  strTitle As String * 25
  strType As String * 5
End Type
```

4.2.11 STATEMENT CONST – UNCHANGED VALUE -

The constants are fixed, predefined numbers or strings. The contents of the constants do not change during the duration of the procedure. The use of constants allows easier handling of values. They also lead to a better readability of a macro. They are defined directly at the beginning of the procedure with the **Const** statement. Then follow the name and the value assignment.

Variables can also be used instead of constants, but our macro becomes too confusing. With**Const** we say that this definition is not changeable!

Download free eBooks at bookboon.com

```
Sub Constants_1()
  'VAT
  Const VAT As Single = 0.19
  'Filename
  Const Filename As String = "Constant.xlsx"
  Const strPath As String = "C:\Harun\"
  Const strPath_all As String = "C:\Harun\Constant.xlsx"
   'Dim's
  Dim Value_1 As Integer
  Dim Text_1 As String
   Value_1 = 2321
   Text_1 = "Text Text"
       Debug.Print "VAT is: " & vbCr & VAT & "%" & Chr(10)
       Debug.Print "File name is: " & Chr(13) & Dateiname & vbLf
       Debug.Print "Path is: " & vbCr & Pfad & Chr(10)
       Debug.Print "Path & file name is: " & vbCr & strPath_all & vbLf
       Debug.Print "Path & file name is: " & vbCr & strPath & filename & vbLf
       Debug.Print "Value is: " & Chr(13) & Value_1 & vbLf
       Debug.Print "Text is: " & Chr(13) & Text_1 & vbLf
End Sub
```

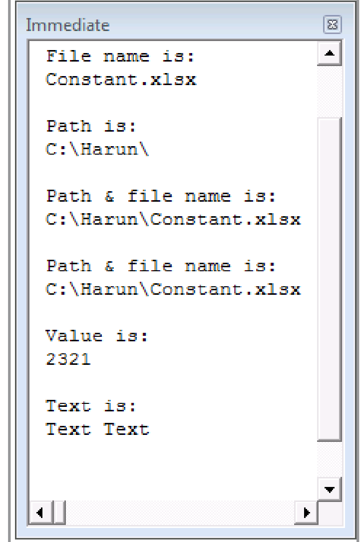


Figure 42:Result of Constants_1

If we write like this below, we will get an error message. A const value must not be changed. This procedure works with **Dim** VAT **As Integer**:

```
Sub Constants_2() '
 Const VAT As Single = 0.19
   VAT=VAT + 1
   MsgBox VAT
End Sub
```

4.2.12 VBA CONSTANTS

There are also many (system) integrated constants in VBA. They begin with the prefix

- ➪ **vb** Visual Basic
- ➪ **xl** Excel.

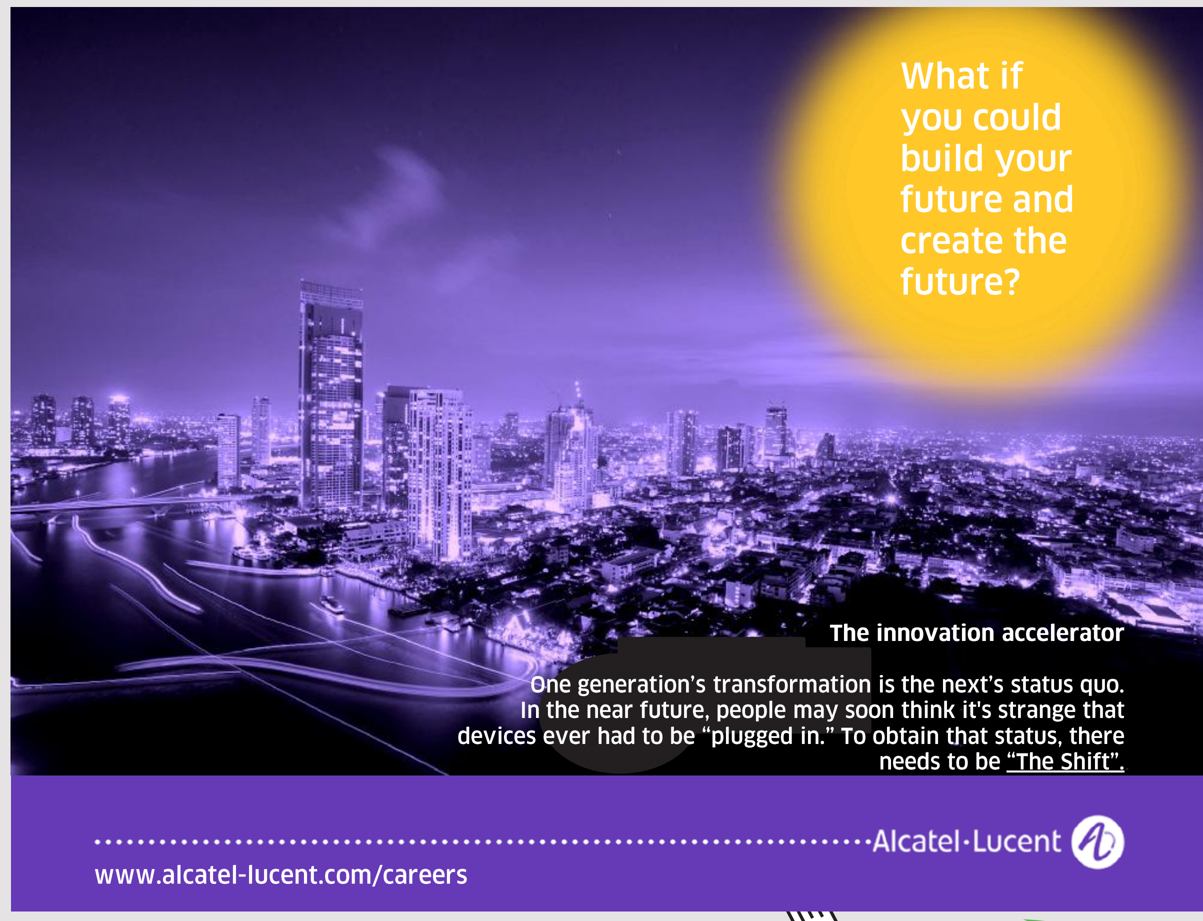
For example, the following color constants can be used directly throughout the code:

Constants	Description
vbBlack	Black color
vbRed	Red color
VbGreen	Green color
vbYellow	Yellow color
vbBlue	Blue color
vbMagenta	Magenta color
vbCyan	Cyan-blue color
vbWhite	White color

Table 2: VBA constants

In this example, the active cell is displayed with a green background color.

```
Sub VBA_Constant()
  ActiveCell.Interior.Color = vbGreen
End Sub
```



4.3 ARITHMETIC-, COMPARISON- UND LOGICAL OPERATORS

The values for the arithmetic process can come from different data sources. For instance:

- as a variable in the program
- from Excel sheet
- as an intermediate value from a calculation
- a result from a formula

4.3.1 ARITHMETIC OPERATORS

These serve to carry out calculations. My first example will be with the values from the examples mentioned above.

```
Sub Addition_Subtraction()
Dim x, y, z, xy, xyz As Integer
Dim a, b, ab As Single

x = 234
y = -123

a = Range("A1").Value '12,5
b = Range("B1").Value '25,7

z = (x - y) + 35

Debug.Print "From Variable: " & Chr(10) & x + y
Debug.Print "From Excel-Sheet: " & Chr(10) & a + b
Debug.Print "From Formula: " & Chr(10) & z
End Sub
```

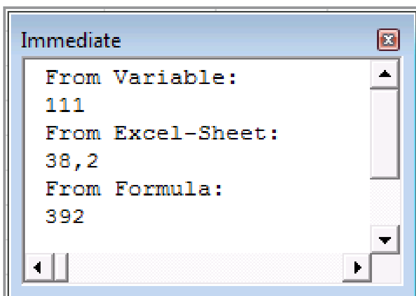


Figure 43: Example arithmetic operators

Download free eBooks at bookboon.com

The following arithmetic operators are used in the VBA code:

*	Multiplication
/	Division
\	Division (Integer, without comma)
^	Potency
Mod	Modulo

The calculate rule says: "Point before line". This means multiplication and division first, then addition and subtraction. This regulation also applies in VBA.

12 + 3 * 5 = 27

Other orders may need to be specified by parentheses:
(12 + 3) * 5 = 75

4.3.2 COMPARISON OPERATORS

Comparison operators can be applied to almost all data types. The result of the comparison is a truth value - either

⇨ **TRUE ➔ correct,**
⇨ **FALSE ➔ not correct**.

Comparison operations can be performed in the VBA using the following operators:

Operator	Description
<	less than
<=	less than or equal to
>	more than
>=	more than or equal to
=	equal
<>	unequal
Is	compare an object
Like	compare a pattern

Well, here is an example of IS or LIKE:

Download free eBooks at bookboon.com

- IS is used to compare two different variables with references to object.

```
Sub IS_Compare()
Dim Object1, NewObject, Object2, Object3, Object4, Test1
Set NewObject = Object1          ' Object allocation
Set Object2 = Object1
Set Object4 = Object3
 Test1 = NewObject Is Object2    ' result true.
 Test1 = Object4 Is Object2      ' result false.
' Accepted: Object1 <> Object3
 Test1 = Object1 Is Object4      ' result False.
End Sub
```

If both Object1 and Object2 refer to the same object, the result is TRUE, otherwise the result is FALSE.

- LIKE is used to compare a string with a pattern.

```
Sub like_Compare()
Dim Test1 As String
 Test1 = "aBBBa" Like "a*a"     'First and last letter are true
 Test1 = "F" Like "[A-Z]"        'True. The "F" is content of A-Z
 Test1 = "a2a" Like "a#a"       'True. First and last letter is "a"
 Test1 = "aM5b" Like "a[L-P]#[!c-e]"    'True. First letter is "a" and last letter is content to e.
 Test1 = "BAT123khg" Like "B?T*"    'True. First and third letter are true.
 Test1 = "CAT123khg" Like "B?T*"    'False. First and last letter do not apply.
End Sub
```

4.3.3 LOGICAL OPERATORS

In addition to mathematical and comparison operators, there are logical operators. The result is output with a Boolean expression.

The statement in VBA syntax is:

Result = [*Comparison_1*] **Operator** [*Comparison_2*]

There are six logical operators:

Not performs a logical negation of an expression.
And serves to perform a logical conjunction between two expressions.
Or is used to perform a logical disjunction between two expressions.
Xor is used to perform a logical exclusion between two expressions.
Eqv is used to determine a logical equivalence between expressions.
Imp serves to perform a logical implication between two expressions.

The emergency operator only affects operands, all other logical operators become Linking expressions.

```
Sub Operators()
  Value1 = 1
  Value2 = 5
  Value3 = 10
  Result = Not (Value1 < Value2) 'results False
  Result = (Value2 > Value1) And (Value3 > Value2) results True
  Result = (Value2 < Value1) Or (Value3 < Value2) 'results False
  Result = (Value2 < Value1) Xor (Value3 > Value2) 'results True
  Result = (Value2 < Value1) Eqv (Value3 < Value2) 'results True
  Result = (Value2 > Value1) Imp (Value3 < Value2) 'results False
End Sub
```

Download free eBooks at bookboon.com

5 ERRORS IN VBA

The errors can arise from different sources, for instance:

- Typing error
- missing or incorrect declarations
- infinite loops
- missing constant
- faulty separation of the VBA code line
- unauthorized / unexpected input / output or
- incorrect entry in the VBA line, etc.

In case of unauthorized or unexpected input, a runtime error appears with an error message and error code. "**Runtime error index out of range**".

There are also different reactions to these error messages. For example, in typing errors the whole VBA code line will appear in red.

To avoid such error messages, the "**Option Explicit**" instruction is helpful. It is written at the beginning in each module level. But then we have to declare each variable in the procedure until all are declared. That could be a bit tedious in the beginning, but it helps a lot.

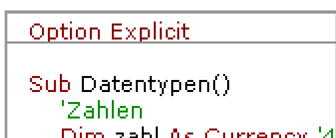
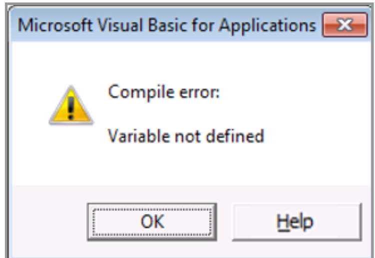


Figure 44: Option Explicit

5.1 ERRORS EXAMPLES




```
Option Explicit
Sub Missing_Variable()
'Variable "NewMessage" is missing
  MsgBox (NewMessage)
End Sub
```

Figure 45: Missing Variable Figure

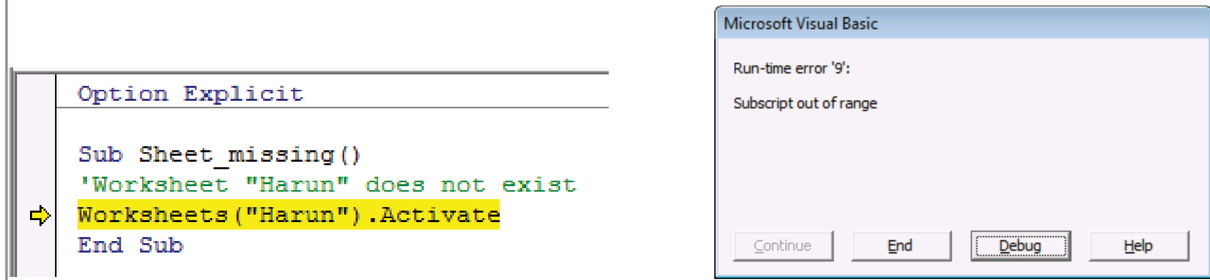


Figure 46: incorrect / missing Definition

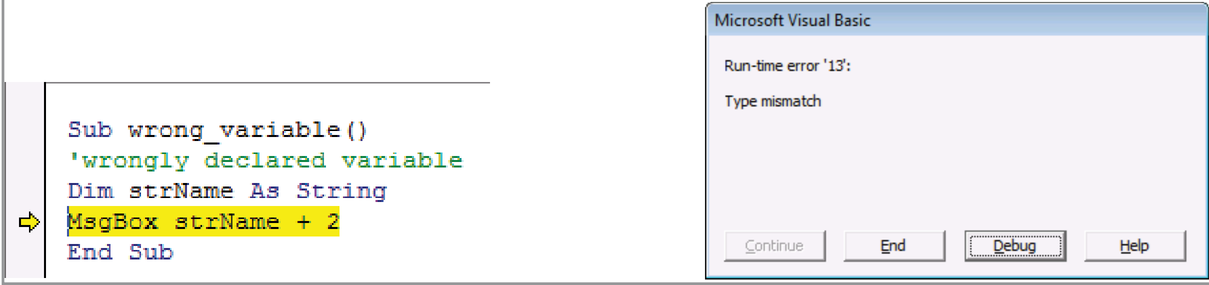


Figure 47: incorrect / missing input or output

The information in the error message is unfortunately not very useful. With the button "Debug" we get to the "faulty" VBA-code line which is highlighted in yellow. With the "Help" button we get to the help text if something and understandable entered. This does not usually help us find a solution. Unfortunately, we have to live with that. The "Finish" button terminates or aborts the procedure depending on the error message.

5.2 HANDLING ERRORS

The error messages can be suppressed with the **On Error Resume Next** instruction. From the "On Error Resume Next" line, all error messages are suppressed. It is valid in the whole procedure. It can't be recognized by fatal errors.

If necessary, On Error Resume Next can be deactivated with**On Error Goto 0**.
On Error Goto *VBA-String line* redirects the VBA code flow to the line.

In our example you will find two tables, "Harun" and "Kaplan", which are not present. The first mistake is corrected. In the table "Kaplan", our error message appears again.

Download free eBooks at bookboon.com

```
Sub Worksheet_not_exist_2()
'Worksheet "Harun" does not exist!
On Error Resume Next
Worksheets("Harun").Activate

On Error Goto 0
'Worksheet "Harun" does not exist, too!
Worksheets("Kaplan").Activate
End Sub
```

But we can also redirect the process to a message in case of an error:

```
Sub Worksheet_does_not_exist3()
'Worksheet "Harun" does not exist!
On Error GoTo Meldung
Worksheets("Harun").Activate
'
'other VBA-Code line
'
Meldung:
MsgBox " In case of error, I will be redirected here!"
End Sub
```

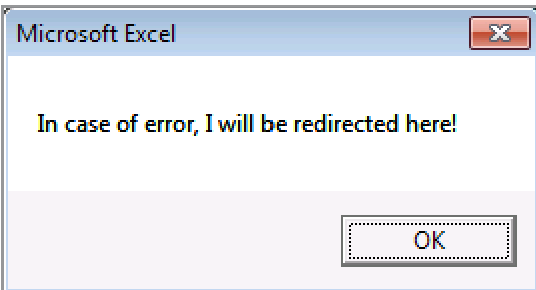


Figure 48: Redirect of error

5.3 TOOLBAR DEBUG

Debugging is an important part of VBA. It is used primarily for tracking variable content. Or in case of an error message, VBA suggests starting the debugger.

Figure 49: Toolbar Debug

Now let's have a look at the individual icons with their tasks.

5.3.1 STEP INTO

The macro can be executed in single step mode also named "Step by Step".

Actions of the function keys:

- executed step by step with the F8 key,
- executed with the key F5 to the end,
- aborted with the key combination Ctrl + Shift + F8,
- repeated with the key combination Ctrl + F8,
- Position the yellow arrow on any line with the mouse and continue from there with the F8 or F5 keys.

The current line is marked on the left bar with a yellow arrow and the line is highlighted in yellow. If you want to know the contents of the variable, just move the mouse pointer over the variable. A tooltip displays the content of the variable.

In our example it looks like this

```
Sub Monitoring()
Dim i As Integer
    For i = 1 To 10
        Debug.Print i
    Next i
End Sub
```

i = 4

Figure 50: Current line in VBA-Editor

5.3.2 CURRENT LINE

When debugging in a single step, the current line is highlighted in yellow and marked with a yellow arrow in the bar to the left of the code.

5.3.3 TOGGLE BREAKPOINT

The Toggle breakpoint serves to stop the drain at this point. The activated line is marked with a red dot. The line is saved with the same color.

The program automatically runs through to the breakpoint and stops at the breakpoint.

Download free eBooks at bookboon.com

The breakpoints can:

- Be inserted or removed on the desired line with the **F9** key.
- Be inserted or removed on the desired line with a mouse click.
- remove all breakpoints with the key combination **Ctrl + Shift + F9**.
- Be carried out automatically with the **F5** key until the next breakpoint.

5.3.4 BOOKMARKS

In order to be able to reach certain places more quickly, so-called bookmarks can be inserted on the left bar. We go to the line and click on the flag icon "Bookmark" in the toolbar "Edit". A square appears with rounded corners on the bar.
The bookmark can be used multiple times.
The same icon is also used to disable individual bookmarks. You can jump between the bookmarks with the other two flag symbols

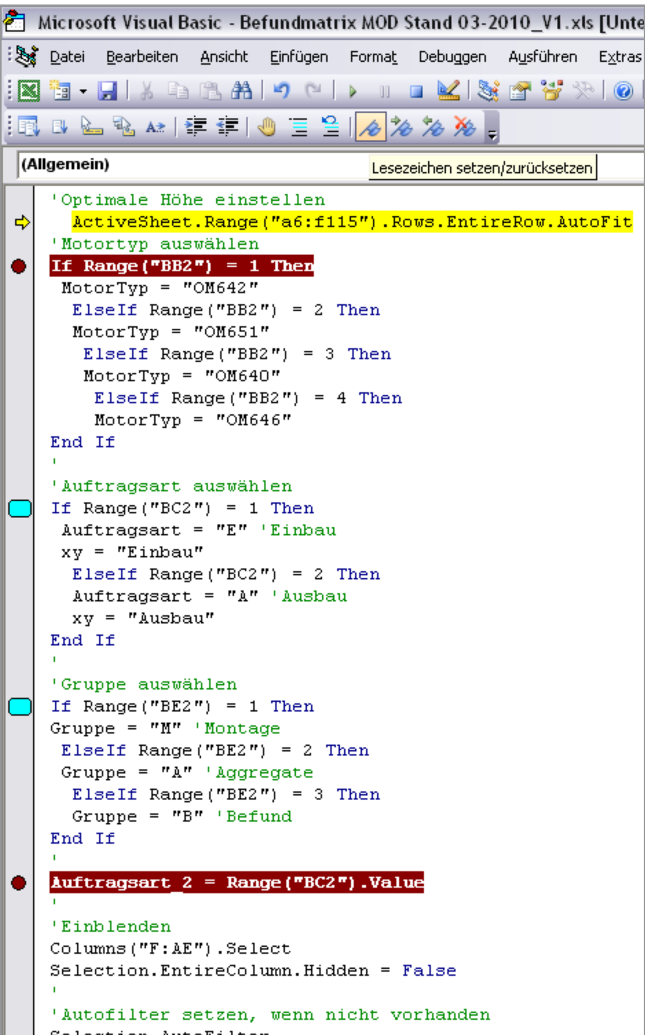


Figure 51: Example bookmarks

6 PROGRAM SEQUENCE, BRANCH UND LOOPS

An important VBA code part is the control of the program sequences depending on the situation. The procedure should react differently depending on the determined value. It's like a traffic light. If green then drive and if red then stop, or if yellow then yield.

Or in other words: as long as nothing changes, this action is carried out; as soon as something changes, the action is also changes.

6.1 DECISION STRUCTURE, IF QUERY, BRANCHES AND RETURNS

Branches and jump labels are used to change the program sequence due to certain conditions or to jump to a designated place and continue from there.

A query works after an operation with the following program section. If the condition is true, this section will be processed and if not, another section will be processed.

In VBA there are the following statements to realize such queries:

- IIF function
- Only useful for a query / decision.
- If-Then-Else statement; completed with End If
- For clarity, this variant is for max. three queries / decisions makes sense.
- Select Case statement; completed with End Select.

6.1.1 IIF FUNCTION

For simpler decisions, there is the Iif decision. If we have only one decision, we can easily query them with the IIf function. Our example may look like this:

```
Sub Movie ticket_with_Iif()
   Note = InputBox("Well, what grade did you get?")
   MsgBox IIf(Note > 2 And Note < 3, " Good, here is your movie ticket. Have fun!", "Sorry.")
End Sub
```

Download free eBooks at bookboon.com

6.1.2 IF ... THEN ... ELSE STATEMENT

A commonly used decision is if ... Then ... Else. We know this process from Excel as an if-then function. How does it work? They work like a vegetable and fruit shop. You get a new delivery. You should then put the goods on the right shelf. If you have a box of apples in your hand, put it on the fruit shelf. If you have a box of lettuce in your hand, put it on the vegetable shelf.

In VBA we can write them with a decision in long form or in short form:

Langform:

```
 If grade <= 3 Then
    MsgBox "Good, here is your movie ticket. Have fun."
End If
```

Shortform:

```
If grade <= 3 Then MsgBox "Good, here is your movie ticket. Have fun."
```

An individual query is rarely used in practice. Most queries are at least two, three, or more. Now let's look at simple to complex examples.

For example, suppose you want to give your child a movie ticket, but it depends on his/her grades. If he/she receives better than a 3, then your child will get the movie ticket. If they receive lower than a 3, they will not.

VBA code might look like this:

```
Sub Movie_Ticket()
grade=InputBox("Well, what grade did you get?")

   If grade <= 3 Then
     MsgBox "Good, here is your movie ticket. Have fun."
   Else
     MsgBox "Sorry."
   End If
End Sub
```

What if there is more decision-making criteria? Now several If..Then's are among themselves. This variant is not recommended because it will require a little more typing. Nevertheless, we can look at this example:

```
Sub if_statement_2()
grade =InputBox("Well, what grade did you get?")
   If grade > 1 And Note <2 Then
     MsgBox "Great. Here is your movie ticket with popcorn and ice cream. Have fun."
      Elseif grade > 2 And Note < 3 Then
       MsgBox "Great. Here is your movie ticket with popcorn. Have fun."
         ElseIf grade < 4 Then
           MsgBox "Good, here is your movie ticket. Have fun."
   Else
     MsgBox "You should still learn. Sorry."
  End If
End Sub
```

We can stack them like a matryoshka nesting doll. Compared to the previous example, there is a little less typing. Now the same example stacked with Elseif:

```
Sub if_statement_1()
grade =InputBox("Well, what grade did you get?")
   If grade = 1 Then
     MsgBox "Great. Here is your movie ticket with popcorn and ice cream. Have fun."
   End If
   If grade = 2 Then
     MsgBox "Great. Here is your movie ticket with popcorn. Have fun."
   Endif
   If grade = 3 Then
    MsgBox "Good, here is your movie ticket. Have fun."
   Endif
   If grade > 3 Then
     MsgBox "You should still learn. Sorry!"
   End If
End Sub
```

The If statement has checked fixed notes on previous examples. If entered with a number, no correct message would appear. Now we have made our example even more accurate. Here we have also given the notes between 1 and 2, etc.:

```
Sub if_statement_with_Elseif()
grade =InputBox("Well, what grade did you get?")
   If grade = 1 Then
    MsgBox "Great. Here is your movie ticket with popcorn and ice cream. Have fun. "
      Elseif grade = 2 Then
        MsgBox "Great. Here is your movie ticket with popcorn. Have fun."
         ElseIf grade = 3 Then
           MsgBox "Good, here is your movie ticket. Have fun."
   Else
     MsgBox "You should still learn. Sorry."
  End If
End Sub
```

The next example checks whether the input corresponds to a value, i.e. no letter or similar. We check this with **IsNumeric**:

```
Sub if_statement_3()
grade =InputBox("Well, what grade did you get?")
If IsNumeric(Note) Then
   If grade > 1 And Note <2 Then
     MsgBox "Great. Here is your movie ticket with popcorn and ice cream. Have fun."
       Elseif grade > 2 And Note < 3 Then
         MsgBox "Great. Here is your movie ticket with popcorn. Have fun."
           ElseIf grade < 4 Then
            MsgBox "Good, here is your movie ticket. Have fun."
    Else
      MsgBox "You should still learn. Sorry."
   End If
  Else
   MsgBox "Please enter your grade!!"
End If
End Sub
```

Now a practical example. We check the contents of a current cell. Depending on the content of the cell, it should output what the content is and has. The output should be output according to the type.

If the content contains a number, then a message with cell content appears. Otherwise, if the content contains a date, then another message with cell content appears, etc.

```
Sub Check_cell_value_1()
    If IsNumeric(ActiveCell) And ActiveCell <> "" Then
        MsgBox  "The cell value is a number! " & ActiveCell.Value
          ElseIf IsDate(ActiveCell) Then
            MsgBox  "The cell value is a date" & ActiveCell.Value
               ElseIf ActiveCell <> "" Then
                MsgBox  "The cell value is a Text!" & ActiveCell.Value
                  ElseIf ActiveCell = "" Then
                    MsgBox "The current cell is empty!"
    End If
End Sub
```

With a refinement it may look like this:

We divide the text to a variable "content". The MsgBox statement, as seen several times, issues this message. It is written at the end of the listing with the variable "content" AND with active cell content.

Firstly, this is clearer and secondly, it is more adaptable or adjustable.

Download free eBooks at bookboon.com

I think it looks a bit clearer.

```
Sub Check_cell_value_2()
    If IsNumeric(ActiveCell) And ActiveCell <> "" Then
        Inhalt =  "The cell value is a number!"
        ElseIf IsDate(ActiveCell) Then
           Inhalt = "The cell value is a date!"
           ElseIf ActiveCell <> "" Then
             Inhalt = "The cell value is a text!"
             ElseIf ActiveCell = "" Then
               Inhalt = "The current cell is empty!"
    End If
        MsgBox Inhalt  & ActiveCell.Value
End Sub
```

Another practical example:

In this example, the input is checked by number. If there is no number, then a message with "Enter number" and macro will be terminated. If the input is a number, it is checked, then, depending on the size of the input, the variables "a" and "b" are assigned values. These will be calculated and spent at the end accordingly.

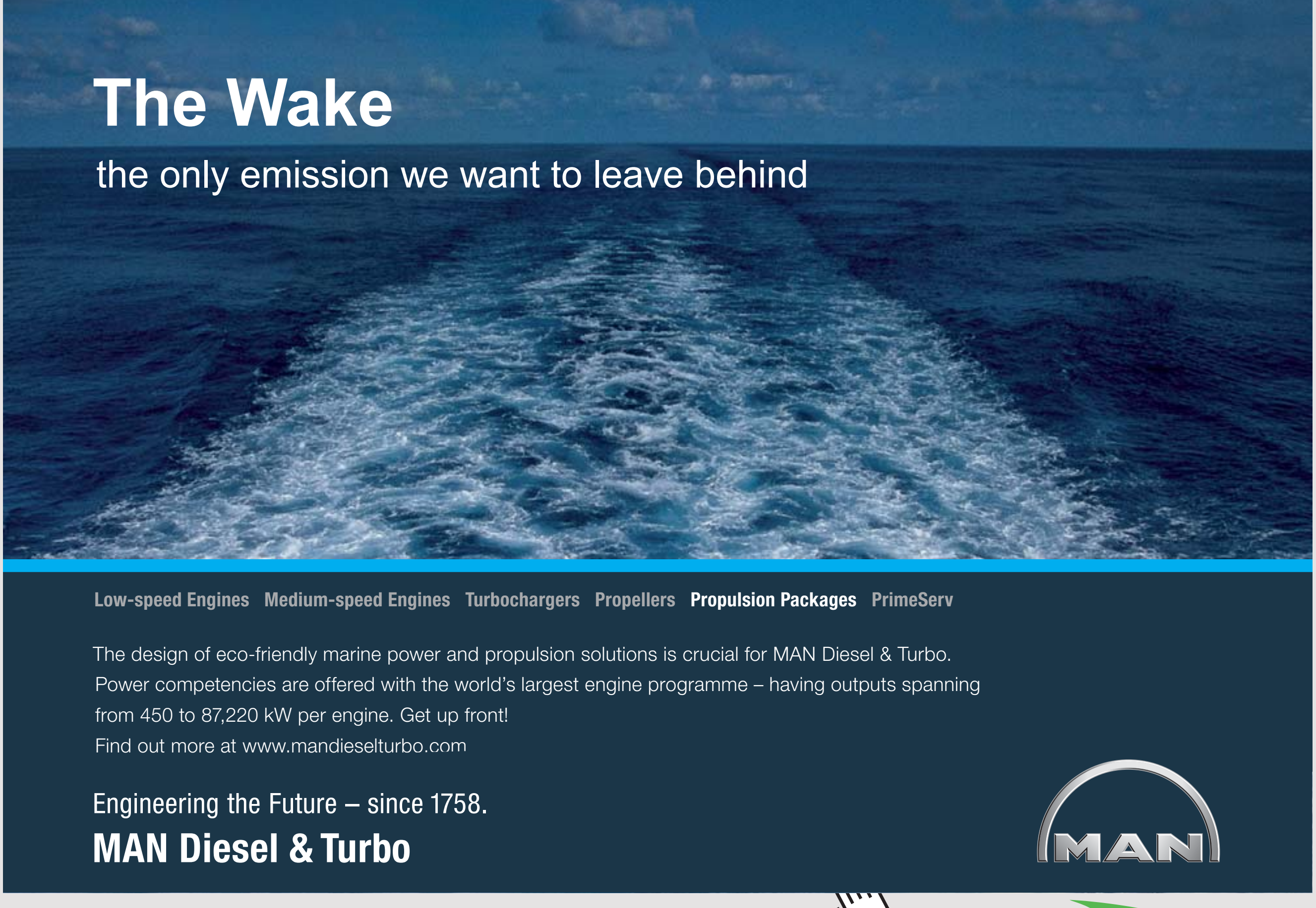
```
Sub Practical_Example()
    intInput = InputBox("Please enter your number!")
    If IsNumeric(intInput) = True Then
        If intInput < 5 Then
           a = 2
           b = 3
            ElseIf intInput > 5 And intInput < 10 Then
                a = 4
                b = 5
            ElseIf intInput > 10 And intInput < 15 Then
                a = 6
                b = 7
        End If
     Else
     MsgBox ("Please enter your number!")
     Exit Sub
     End If
    intResult = a * b
    MsgBox intResult
End Sub
```

Download free eBooks at bookboon.com

6.1.3 SELECT CASE-; END SELECT STATEMENT

If multiple inputs or values need to be validated and another event is to occur then "if statements" may become cluttered. In this case we need precise instruction. This is the Select Case statement. Here the content of a variable is checked for a possible match.

```
Sub Select_Case_statement_I()
grade=InputBox("Well, what grade did you get?")
  Select Case grade
   Case 1
     MsgBox "Great. Here is your movie ticket with popcorn and ice cream. Have fun!"
   Case 2
     MsgBox "Great. Here is your movie ticket with popcorn. Have fun!"
   Case 3
     MsgBox "Good, here is your movie ticket. Have fun!"
   Case 4
     MsgBox "Alright. Exceptionally done. Here is your movie ticket! "
   Case >= 5
     MsgBox "You should still learn. Sorry! "
   End Select
End Sub
```



Click on the ad to read more

And here with comma input:

```
Sub Select_Case_Anweisung_II()
grade =InputBox("Well, what grade did you get?")
  Select Case grade
   Case 1 To 2
     MsgBox "Great. Here is your movie ticket with popcorn and ice cream. Have fun!"
   Case 3 To 4
     MsgBox "Good, here is your movie ticket. Have fun!"
   Case >= 4
     MsgBox "You should still learn. Sorry! "
   Case ""
     MsgBox "If you do not enter a grade, you will not get a movie ticket!"
  End Select
End Sub
```

6.2 LOOPS

A loop is a framework which repeats a certain part of the program several times. The loop part in a procedure is as specified unless another condition is fulfilled. A distinction is made between pre- and post-tested loops. That is, whether the check condition is in the loop head or loop foot of the repeat mechanism.

The loop can be terminated prematurely when entering certain state with the exit statement.

Again, there are different loop instructions:

- For...Next statement
- While…Wend statement
- Do...Loop statement

Now let's take a look at them with examples:

6.2.1 FOR ... NEXT – STATEMENT

For...Next - A statement is usually used in combination with a counter that increments after each loop. The counter used here, e.g. "I", tells you how many times the loop should go through. The condition is defined in the head of the loop.

In the following example, the counter "i" through 8 is run through. After each run, the counter increases by the value 1. So i = i + 1.

Download free eBooks at bookboon.com

```
Sub forward_loop()
 For i = 1 To 8
  Debug.Print i
 Next i
End Sub
```

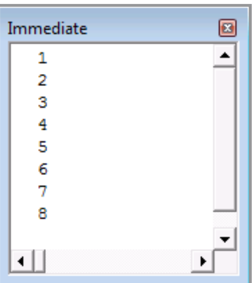


Figure 52: For...Next loop forward

What if the counter should always be increased by the value 2? So, i = i + 2. We'll do that with the step statement. Thus we indicate with which step the counter should be increased. If this jump has no sign (plus or minus) in front of the digit, it is always forward or positive.

```
Sub backward_loop_step_2()
 For i = 2 To 12 Step 2
  Debug.Print i
 Next i
End Sub
```

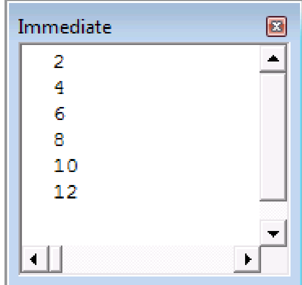


Figure 53: For...Next loop backward

In the following example we let the counter "i" go through this way 8 times. After each run, the counter becomes 1 less. So, i = i – 1.

```
Sub backward_loop()
 For i = 8 To 1 Step -1
  Debug.Print i
 Next i
End Sub
```

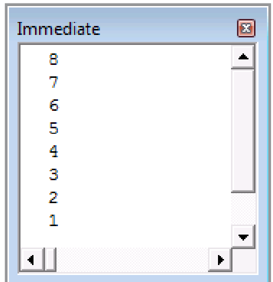


Figure 54: For...Next loop backward

Now a combination with If...Then...Else decisions. The run in the lower example runs with three steps backward and is conditionally terminated prematurely. Once the value i is less than 3, exit with Exit For before appearing in the immediate area. So, i = i – 3.

```
Sub Combination_If_and_For()
  For i = 20 To 1 Step -3
    If i<3 Then Exit For
    Debug.Print i
  Next i
End Sub
```

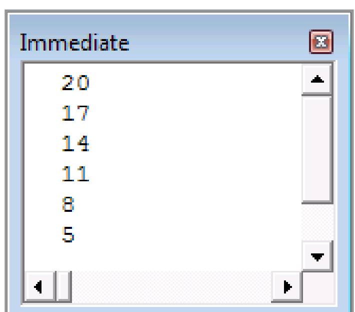


Figure 55: Combination For...Next / If...Then

Here is a practical example- The cell contents of a table are read out and output. With the Cells / cell directive, we read the contents of the cell as text. For example, if a cell contained "123", it would be read as text. But if you want to read it as a number, value / value would have to be written. We will see more of that later.

The first value in parenthesis of the Cells statement indicates the row and the second value indicates the column. In our example, "x" is incremented by "1" after each pass, starting at "1". The second value does not change, it remains at "1", i.e. the column "A". In addition, we are rebuilding the If...Then...Else decision here. Once the cell to be read is empty, our loop is terminated.

```
Sub Select_Value()
For x = 1 To 15
  Debug.Print Cells(x, 1).Text
    If Cells(x, 1).Text = "" Then Exit For
 Next x
End Sub
```

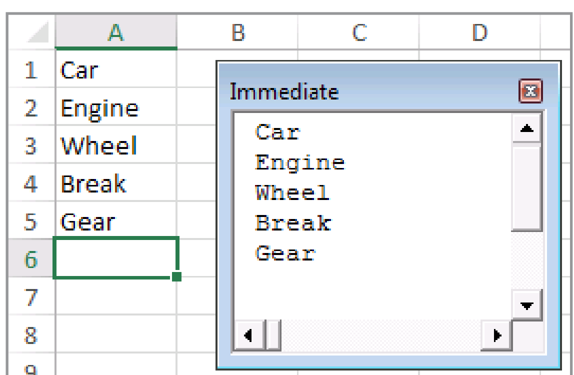


Figure 56: Select value from sheet

Now I just want to show a part of the practical example. We will come back to these topics later. Here is an example to make sense of the For-Next statement.
This is about controlling special characters in a string. We assume that the contents of a cell contain this string. Our example controls until a special character is found.
Our loop starts at "length = 1" and runs maximum number of "Len (cell_content)" through. As soon as a special character occurs, exit with "Exit Sub".

Download free eBooks at bookboon.com

```
For length = 1 To Len(cell_content)
  Select Case Mid(cell_content, length, 1)
    Case "\", "/", ":", "*", ">", "<", "[", "]"  'List of special character
      MsgBox "The cell contain special characters." & Chr(10) _
      & Mid(cell_content, length, 1) _
      & Chr(10) & "Restart after the correction."
    Exit Sub
  End Select
Next
```

6.2.2 FOR EACH ... NEXT STATEMENT

For Each...Next - statement is mostly used on objects. These objects may be cells, spreadsheets etc.

This loop repeats instructions for all elements of a data field with the number of passes equaling the number of elements.

I think two examples suffice. We will use For Each..Next statement a bit more later on.

In the example below, all sheets (tables, charts, etc.) are output.

```
Sub All_Sheets_Name()
    Dim objRegister As Object
    For Each objRegister In ThisWorkbook.Sheets
        Debug.Print objRegister.Name
    Next objRegister
End Sub
```

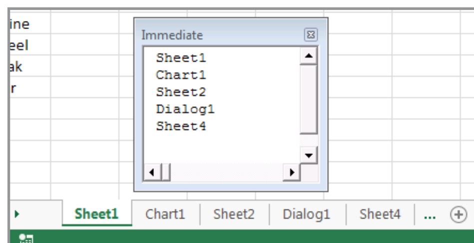


Figure 57: Names of all sheets

In the example below, all worksheets are output.

```
Sub Only_Table_Sheets()
    Dim wsSheet As Worksheet
    For Each wsSheet In WorkSheets
        Debug.Print wsSheet.Name
    Next wsSheet
End Sub
```

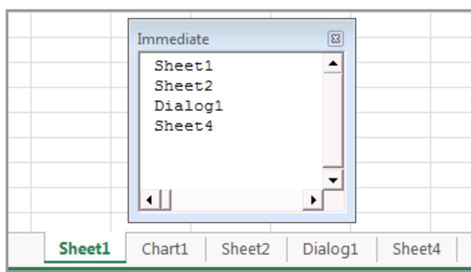


Figure 58: Names of Sheets only

Download free eBooks at bookboon.com

6.2.3 DO…LOOP STATEMENT

There are two types of von Do…Loop statement:

- **While**
  Checking is carried out in advance.

- **Until**
  Checking will be carried out afterwards.

In the example below we are looking for the term "wheel" with Do ... While ... Loop. In column "A" it is searched until the entry "Wheel" is found. As a result, the address of the cell is output.

If the contents are incorrect, cell contents are output and if the result is correct, the address of the cell is output from the searched term

```
Sub DoWhileLoop_Statement_1()
 i = 1
  Do While Range("A" & i) <> "Wheel"
    Debug.Print Range("A" & i)
    i = i + 1
  Loop
    Debug.Print Range("A" & i). Value & "Found in cell: " & Range("A" & i).Address
End Sub
```

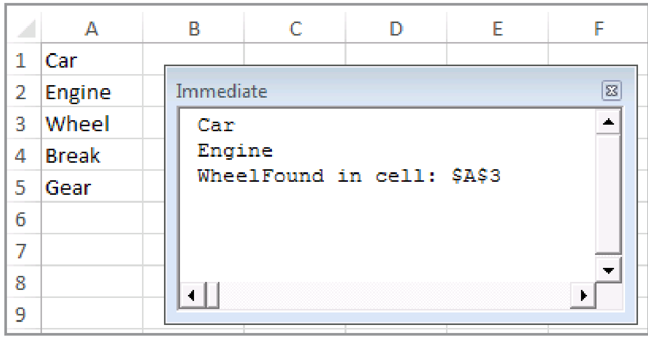


Figure 59: Search with Do…Loop_1

It is searched until an empty cell is found.

```
Sub DoWhileLoop_statement_2()
 i = 1
   Do While Range("A" & i) <> ""
     Debug.Print Range("A" & i)
      i = i + 1
   Loop
     Debug.Print "Next empty cell is in: " & Range("A" & i).Address
End Sub
```

Download free eBooks at bookboon.com

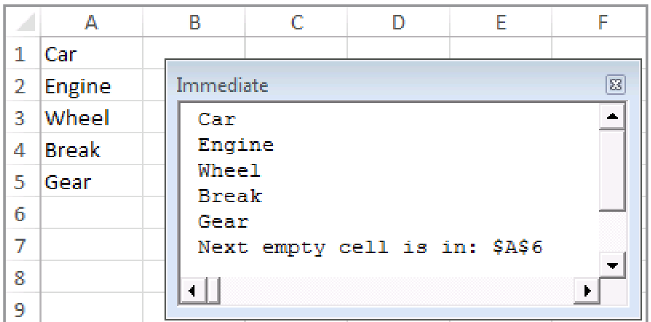


Figure 60: Suchen_2 Do...Loop

The next two examples are with the Do … Until … Loop statement. The result of this statement is the same as before. The difference next to the Until statement is the operator. Here an equal sign (=) is used instead of "Smaller (<) and Larger (>)" characters.

```
Sub DoUntilLoop_statement_1()
 i = 1
  Do Until Range("A" & i) = "Rad"
    Debug.Print Range("A" & i)
    i = i + 1
  Loop
    Debug.Print "Found in cell: " & Range("A" & i).Address
End Sub
```

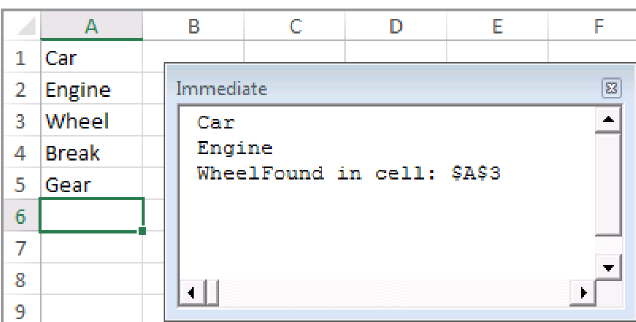


Figure 61: Search with Do...Until_1

This example searches until an empty cell is found.

```
Sub DoUntilLoop_statement_2()
 i = 1
  Do Until Range("A" & i) = ""
    Debug.Print Range("A" & i)
    i = i + 1
  Loop
    Debug.Print "Next empty cell is in: " & Range("A" & i).Address
End Sub
```

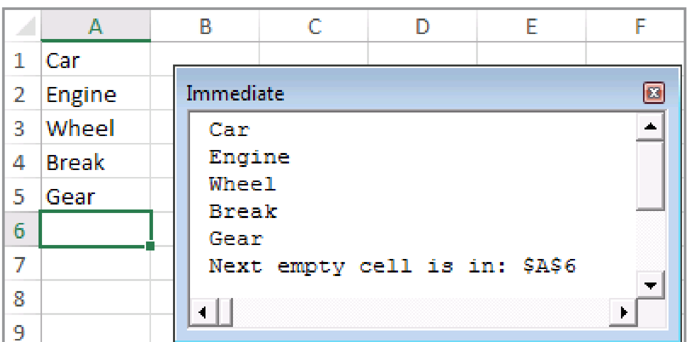


Figure 62: Search with Do...Until_2

6.2.4 WHILE...WEND STATEMENT

While ... Wend statement is executed until the desired operation occurs.

```
Sub WhileWend_statement()
 i = 1
  While Range("A" & i) <> ""
    Debug.Print Range("A" & i)
    i = i + 1
  Wend
    Debug.Print "Next empty cell is in:" & Range("A" & i).Address
End Sub
```

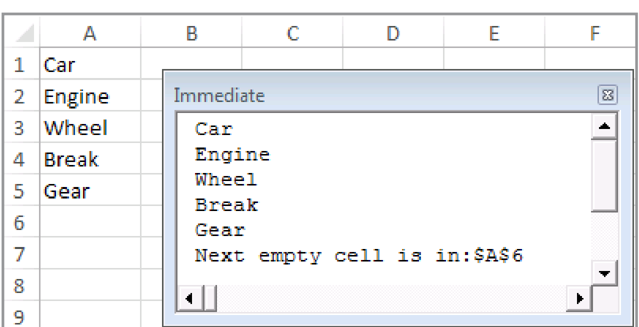


Figure 63: Search with While..Wend

6.2.5 WITH...END WITH STATEMENT

The With statement can combine multiple statements for a given object. It brings us two important benefits:

1. Less paperwork
2. High speed of the module.

Between the With and End With statements we first define "where", then "what" and "how" to display or format. For example, in "cell", "font" should be displayed in bold.

This example demonstrates how a With...End With statement works. Long listings can be seen when macros record. We will see more about macros later on.

On the left is a recorded macro. On the right-hand side is the optimized macro. All items that were not changed during recording are recorded as "False" or "0". These can be easily removed. Stacking the With statements minimizes the listing by a few lines.

```
Sub Recorded_macro()
'
    ActiveCell.FormulaR1C1 = "Harun"
    With Selection.Interior
        .ColorIndex = 34
    End With
    Selection.Font.Bold = True
    Selection.Font.ColorIndex = 5
    With Selection.Font
        .Name = "Arial"
        .Size = 12
        .Strikethrough = False
        .Superscript = False
        .Subscript = False
        .OutlineFont = False
        .Shadow = False
        .Underline = xlUnderlineStyleNone
        .ColorIndex = 5
    End With
    With Selection
        .HorizontalAlignment = xlLeft
        .VerticalAlignment = xlBottom
        .WrapText = False
        .Orientation = 0
        .AddIndent = False
        .IndentLevel = 0
        .ShrinkToFit = False
        .ReadingOrder = xlContext
        .MergeCells = False
    End With
End Sub
```

```
Sub Optimized_macro()
'
    With ActiveCell
        .Interior.ColorIndex = 34
        .Value = "Harun"
        .HorizontalAlignment = xlLeft
        .VerticalAlignment = xlBottom
        .ReadingOrder = xlContext
    `
          With .Font
           .Bold = True
           .Name = "Arial"
           .Size = 12
          .ColorIndex = 5
         End With
    End With
End Sub
```

In the second example, the column and the row of the active cell are set to the optimal height or width.

```
Sub Examples_With_2()
    ActiveSheet.UsedRange.Select
    Selection.Rows.AutoFit
    Selection.Columns.AutoFit
End Sub
```

```
Sub Optimized_Examples_With_2()
    With ActiveSheet.UsedRange
        .Rows.AutoFit
        .Columns.AutoFit
    End With
End Sub
```

6.3 BRANCH STATEMENT

A jump instruction is like a lift in a mall. As soon as you are in the elevator, press the button for the floor you want to go to (decision / jumps). If you are on the floor, continue shopping (program cuts).

Such jumps are executed when a condition occurs or enters directly. There are two types:

- ➪ A **conditional jump**, which is possible after a condition **If..Then..Else** or **Select Case**.

- ➪ An **UNconditional jump,** which continues from the given VBA-code line / jump point, **GoTo** or **GoSub…Return**.

Here are some examples to help make it easier to understand.

6.3.1 GOTO STATEMENT

With this statement, our macro continues from the specified line. This line is terminated with a colon ":". For example, "Note_2_3:" And this line is then justified to the left in our listing.

Download free eBooks at bookboon.com

```
Sub GoTo_Examplel()
grade = InputBox("Well, which grade did you get?")

If grade > 1 And grade < 2 Then GoTo grade_1_2

  Select Case grade
   Case 2 To 3
     GoTo grade_2_3
   Case 4 To 5
     GoTo grade_4_5
   Case Is > 6
     GoTo Ending
   End Select

grade_1_2:
    MsgBox "Great. Here is your movie ticket with popcorn and ice cream. Have fun!"
    GoTo Ending:
grade_2_3:
     MsgBox "Great. Here is your movie ticket. Have fun!"
    GoTo Ending:
garde_4_5:
     MsgBox "You should still learn. Sorry!"
    GoTo Ending:
grade_6:
     MsgBox "You need tutoring!"
Ending:
End Sub
```

6.3.2 GOSUB-; RETURN STATEMENT

It is very similar to the **GoTo** statement. The difference is that the Return statement below the **GoSub** statement returns to where it left off.

I describe it here with a funny example, let's say you have a dog and you two play in a meadow with a ball. You throw the ball to the left, your dog runs over different hills, picks up the ball and brings it back. Now you throw it to the right and the dog runs through the trees and brings the ball back. If you or your dog are tired then you go home. ☺

Here is an example with **GoSub** / **Return** / **Select-Case**:

Download free eBooks at bookboon.com

```
Sub Example_GoSub()
  Range("A1").Select
  i = 0
repeat:
  CellValue = ActiveCell.Offset(i, 0).Value
  ActiveCell.Offset(i + 1, 0).Select
  If CellValue = "" Then Exit Sub
  Select Case CellValue
    Case 1, 2
        GoSub intValue
    Case 3
        GoSub intValue_3
    Case "Gear"
        GoSub Text
  End Select
    GoTo repeat
intValue:
    Factor = 0.5
    Result = CellValue * Factor
    Debug.Print Result
Return
'
intValue_3:
    Factor_3 = 0.7
    Result = CellValue * Factor_3
    Debug.Print Result
Return
'
Text:
    Add_Text = "Additional Text"
    Result = CellValue & vbLf & Add_Text
    Debug.Print Result
Return
'
  If ActiveCell.Offset(i, 0).Value = "" Then
    Exit Sub
  Else
    GoTo repeat
  End If
End Sub
```

Now what is going on here? There are entries in column "A" in the Excel sheet. Depending on the "Content" entry, it is decided which GoSub statement should be executed. If it is"Content = 1,2,4,5" or "6", then "GoSub Number" will be executed. If it is "content = 3", then "GoSub number_3" will be executed, if "content = rt", then "GoSub Text" is executed.

Here's the same example with If-Then

Download free eBooks at bookboon.com

```
Sub Example_GoSub_If()
     Range("A1").Select
     i = 0
 repeat:
   CellValue = ActiveCell.Offset(i, 0).Value
   ActiveCell.Offset(i + 1, 0).Select
   If CellValue = "" Then Exit Sub
   If CellValue = 1 Or CellValue = 2 Then GoSub Zahl
   If CellValue = 3 Then GoSub Zahl_3
   If CellValue = "rt" Then GoSub Text
   GoTo repeat
  intValue:
     Factor_1 = 0.5
     Result = CellValue * Faktor_1
    Debug.Print Result
 Return
 '
 intValue_3:
    Factor_2 = 0.7
    Result = CellValue * Faktor_2
    Debug.Print Result
 Return
 '
 Text:
    Add_Text = "Additional Text"
    Result = CellValue & vbLf & Add_Text
    Debug.Print Result
 Return
 '
 If ActiveCell.Offset(i, 0).Value = "" Then
   Exit Sub
 Else
   GoTo repeat
 End If
End Sub
```

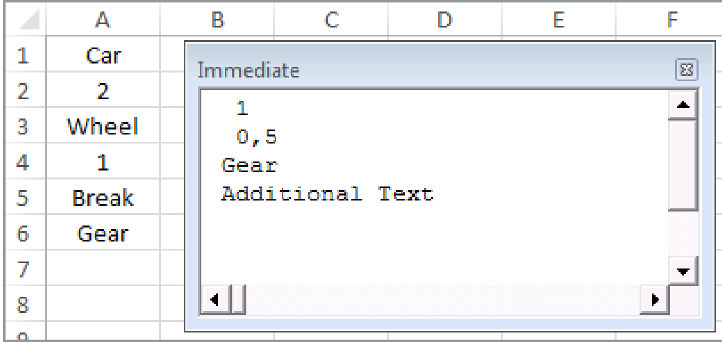


Figure 64: Result of GoSub

Download free eBooks at bookboon.com

7 COMMUNICATION WITH EXCEL

What is communication? The dictionary says:

- Communication between people using language or signs
- The exchange of information between devices

Communication with Excel is a mixture of these two definitions. Because we sit opposite of Excel, we tell the Excel by menu selection what it should do and it follows or it outputs a message window.

We have seen these communication messages very often. For example

- with an error message

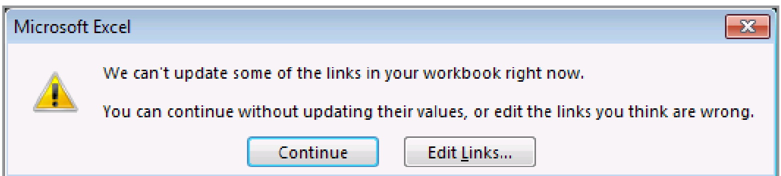


Figure 65: An Error message

- also with the settings in Excel up to version 2010

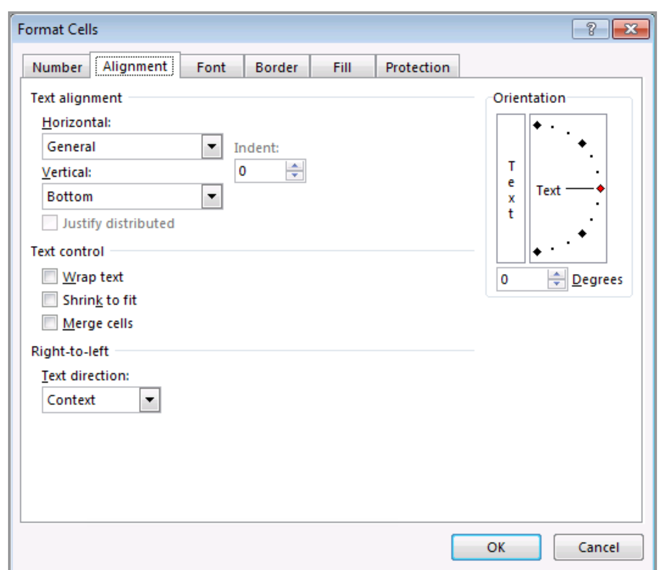


Figure 66: Format Cells

Download free eBooks at bookboon.com

- and also with the settings in Excel from version

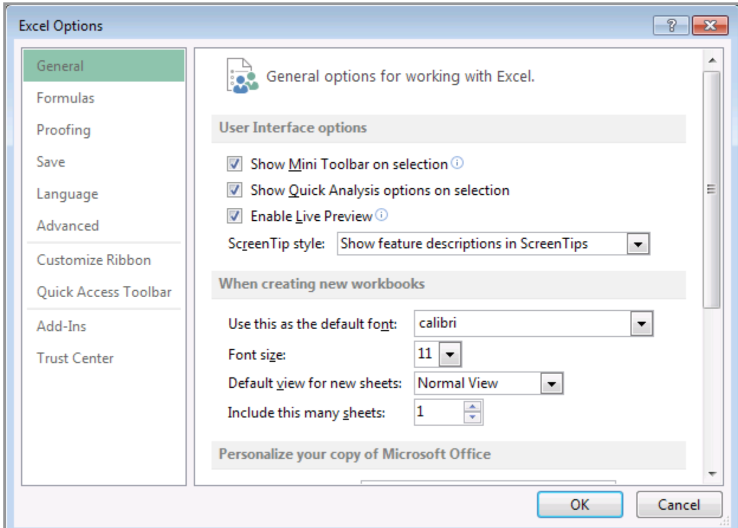


Figure 67: Settings in Excel from Version 2010

Using VBA instructions, we can display such messages or masks in different ways and make them appear on the desktop:

- "InputBox = input field" and "MsgBox = output field"
- Dialog mask "UserForm" (later on in Book IV)

7.1 MESSAGE WINDOW

The message box **MsgBox "Message Box"** is often used in VBA. MsgBox is set to an output message on the screen. MsgBox has at least one "OK" button to confirm. Each button has a return value in the background to distinguish which key is pressed. The selected key then returns a value of type Integer. The message is waiting for a button displayed inside it to be clicked.

The syntax of the MsgBox function with its arguments:
MsgBox (prompt [, buttons] [, title])

As long as a message window is displayed on the screen, you can't work in the Excel file or in the VBE code. The message window must be closed first in order to carry on.

Download free eBooks at bookboon.com

```
Sub MsgBox_simly()
  MsgBox "I learn VBA."
End Sub
```

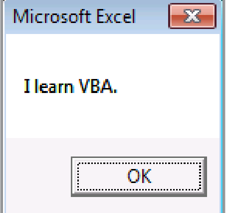
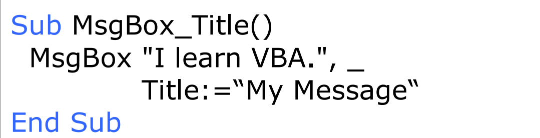


Figure 68: Example MsgBox_1



```
Sub MsgBox_Title()
  MsgBox "I learn VBA.", _
          Title:="My Message"
End Sub
```

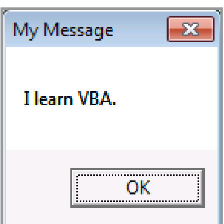


Figure 69: Example MsgBox_2

7.1.1 BUTTONS IN MESSAGE WINDOW

A simple message window has at least one OK button. In addition, other buttons may be displayed. This can be used in VBA code with a constant or with their values. Available buttons are as follows:

Constant	Value	Button
vbOKOnly	0	OK
vbOKCancel	1	OK and Cancel
vbAbortRetryIgnore	2	Cancel, Retry and Ignore
vbYesNoCancel	3	Yes, No und Cancel
vbYesNo	4	Yes and No
vbRetryCancel	5	Retry and Cancel

Table 3: Constant of MsgBox

If you want to display additional buttons other than OK, such as Cancel, either with a constant or the value button then these buttons must be programmed accordingly. In the same section as the "OK" button you will also find "Cancel". In programming, either the If … Then … Else or the Select … Case statement is used. Both variants are displayed below.

Download free eBooks at bookboon.com

First, the example with "OK":

```
Sub MsgBox_1()
  MsgBox "VBA-Programming"
End Sub
```

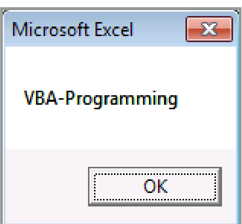


Figure 70: OK Button only

Now "Cancel" comes into play. If there are two buttons, we have to add the clicked button to a value. The example "Msgbox_2_1" with a constant and the example "Msgbox_2_2" with the value of a constant. Both have the same result.

```
Sub MsgBox_2_1()
  Wert=MsgBox ("VBA-Programming", _
          Buttons:=vbOKCancel)
End Sub

Sub MsgBox_2_2()
  Wert=MsgBox ("VBA-Programming", _
          Buttons:=1)
End Sub
```

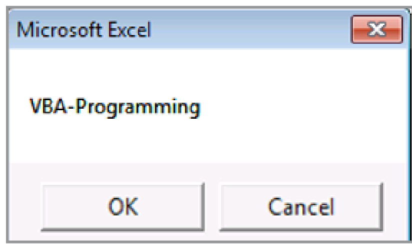


Figure 71: With OK and Cancel buttons

Now with "Cancel", "Repeat" and "Ignore":

```
Sub MsgBox_3_1()
  Wert=MsgBox ("VBA Programming", _
          Buttons:= vbAbortRetryIgnore)
End Sub

Sub MsgBox_3_2()
  Wert=MsgBox ("VBA Programming", _
          Buttons:=2)
End Sub
```

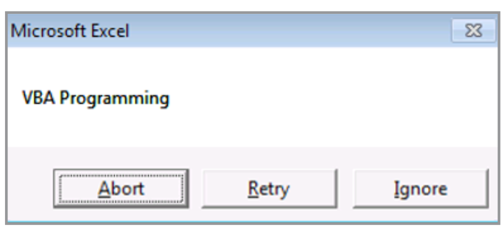


Figure 72: With three buttons

Now we have to program which section should be executed according to the selected button. This is possible with the return value of the selected button. With the "OK" selection we have the "value = 1" and with "Cancel" selection the "value = 2".
The table below lists all the return values.

Constants and (return) values of the individual buttons are as follow:

Download free eBooks at bookboon.com

Constant	Value	Button
vbOK	1	OK
vbCancel	2	Cancel
vbAbort	3	Abort
vbRetry	4	Retry
vbIgnore	5	Ignore
vbYes	6	Yes
vbNo	7	No

Table 4: Return values of a button

Here are two examples. In the first one, we let the OK and Cancel buttons appear in the message window with "**Buttons: = vbOKCancel**". Thereafter, the follow-up action is displayed after the determined return value with the "Select Case" message.

In the second example, we also use "Buttons: = 1" to display OK and Cancel buttons in the message window. The return value is then evaluated with the If..Then decision and a corresponding message is displayed.

Download free eBooks at bookboon.com

```
Sub MsgBox_Button_Select()
 Dim intAnswer As Integer

    intAnswer = MsgBox("I learn VBA.", _
                  Buttons:=vbOKCancel, _
                  Title:="My Message")

 Select Case intAnswer
  Case vbOK
    MsgBox "You have selected OK."
  Case Else
    MsgBox "You have selected Cancel."
 End Select

End Sub
```

```
Sub MsgBox_Button_If()
 Dim intAnswer As Integer

   intAnswer = MsgBox("I learn VBA.", _
              Buttons:=1, _
              Title:="My Message")

   If intAnswer = 1 Then
    MsgBox "You have selected OK."
   Else
    MsgBox "You have selected Cancel."
   End If
End Sub
```

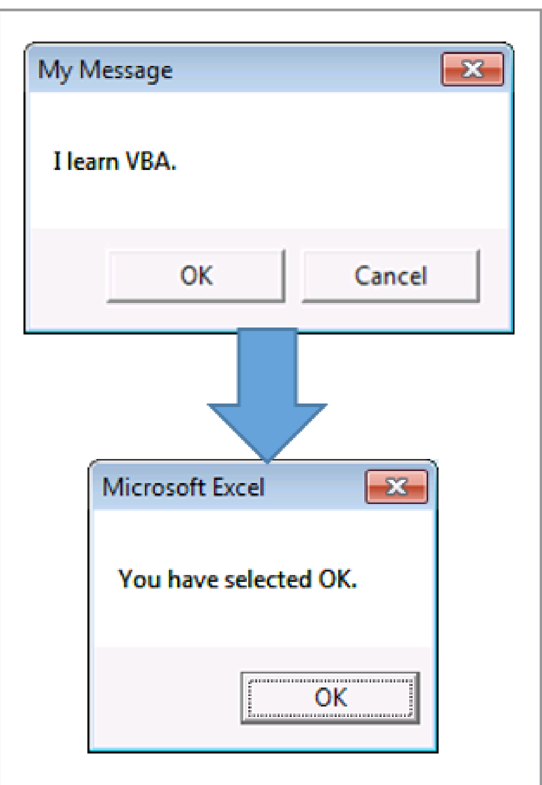


Figure 73: Result from Select Case / If...Then...Else

7.2 BREAK UP OR CONCATENATE MULTILINE TEXT LINES

The width of the message window is a maximum of 128 characters. After that it is automatically wrapped. This can sometimes seem ugly and confusing. It can be formatted better by using the ASCII code and system-internal constants for tab and breaks.

Download free eBooks at bookboon.com

Constant	ASCII	Description
vbTab	Chr(9)	Tabulator
vbLf	Chr(10)	Line Feed – neue Zeile einfügen -
vbCr	Chr(13)	Carriange Return – Zeilenumbruch -

Table 5: Formatting a text in the message window

```
Sub Break_up_or_concatenate()
Dim strFirstName As String

  strFirstName = "Harun,"
'Result in MsgBox
  MsgBox "Hello " & strFirstName & vbLf & vbLf & _
   "Congratulations to your first child." & Chr(10) & _
    Chr(10) & vbCr & _
    Date & vbTab & Time

'Result entered in cell "A1"
 Range("A1").Value = ("Hello " & strFirstName & vbLf & vbLf & _
   "Congratulations to your first child." & Chr(10) & _
    Chr(10) & vbCr & _
    Date & vbTab & Time)
End Sub
```

Figure 74: Output in MsgBox

Figure 75: Result entered in cell

7.3 SYMBOL IN MESSAGE WINDOW

In the message window not only titles, text and buttons can be placed, but additional symbols can be placed as well. This symbol classifies the content of the message, whether the message is a warning, a question, or information, for example.

Download free eBooks at bookboon.com

Constant	Symbol
vbCritical	Stop
vbQuestion	Question mark
vbExclamation	Exclamation mark
vbInformation	Information mark
vbSystemModal	Dialog box remains in the foreground

Table 6: Symbols in Message window

```
Sub MsgBox_Symbol()
MsgBox "That’s a stop mark.", vbCritical
MsgBox "That’s a question mark.", vbQuestion
MsgBox "That’s an exclamation mark.", vbExclamation
MsgBox "That’s an information mark.", vbInformation
End Sub
```

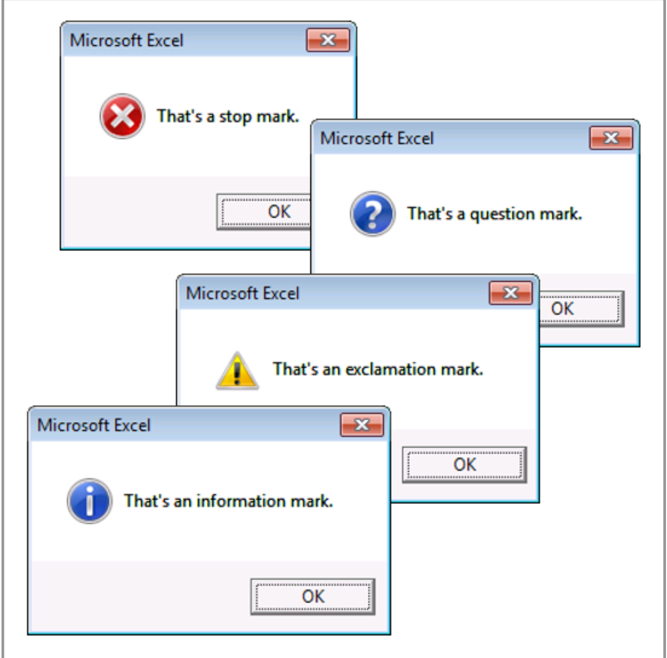


Figure 76: Symbol in Message window

```
Sub MsgBox_SystemModal()
    MsgBox "I stay in sight until you read me." & vbLf & _
        "If you also change your application, I’m still there!", _
        vbOKOnly Or vbSystemModal, "I have to read that!"
End Sub
```

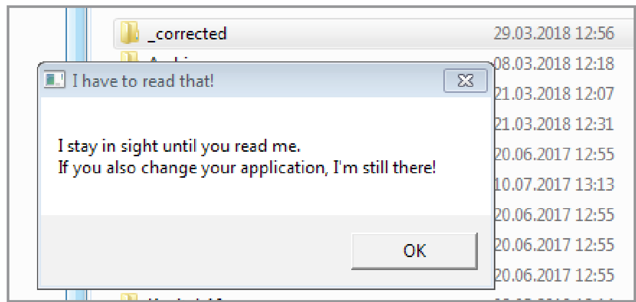


Figure 77: An example with vbSystemModal

The constant **vbSystemModal** causes the message in the foreground to freeze if other programs are also used.

7.4 INPUT WINDOW

The input window, InputBox is the opposite of the output window, MsgBox. As the name implies, this window has an input field and by default contains "OK" and "Cancel" buttons. The select switch then returns a value of type Integer. This value is needed to query which button the user clicked.

The user is prompted

- to enter something
- to click on one of two (OK and Cancel) buttons.

The syntax of the InputBox function with its arguments is in the help text, as follows: **InputBox** (prompt [, title] [, default])

To program the two buttons, either the If ... Then ... Else or the Select ... Case statement is used. The decision is determined according to the constant or the value of the individual buttons.

In the next example, the user is prompted to enter his name. With the "Title" value, the name of the window is entered here "enter name", and with the default value a specific name is entered here "Harun Kaplan". This name is already displayed in the input field.

Download free eBooks at bookboon.com

```
Sub Example_InputOutput()
Dim strmy_Name As String

strmy_Name = InputBox _
("Please enter your name: ", _
                Title:="Enter your name", _
                Default:="Harun Kaplan")
MsgBox strmy_Name
End Sub
```

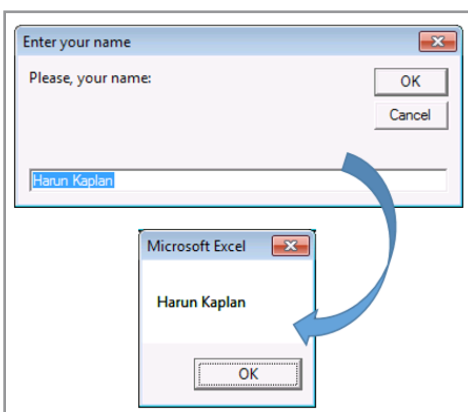


Figure 78: InputBox and MsgBox

```
Sub Message_area_Inputbox() 'with Inputbox
Dim strName, strFirstname, strAddress, strInfo As String
  strName = InputBox("Please enter your name: ")
  strFirstname = InputBox("Please enter your first name: ")
  strAddress = InputBox("Please enter your street: ")
  strInfo = MsgBox(("Name : " & strName & vbCr & _
                      "First name : " & strFirstname & vbCr & _
                      "Address : " & strAddress), , "Information")
End Sub
```

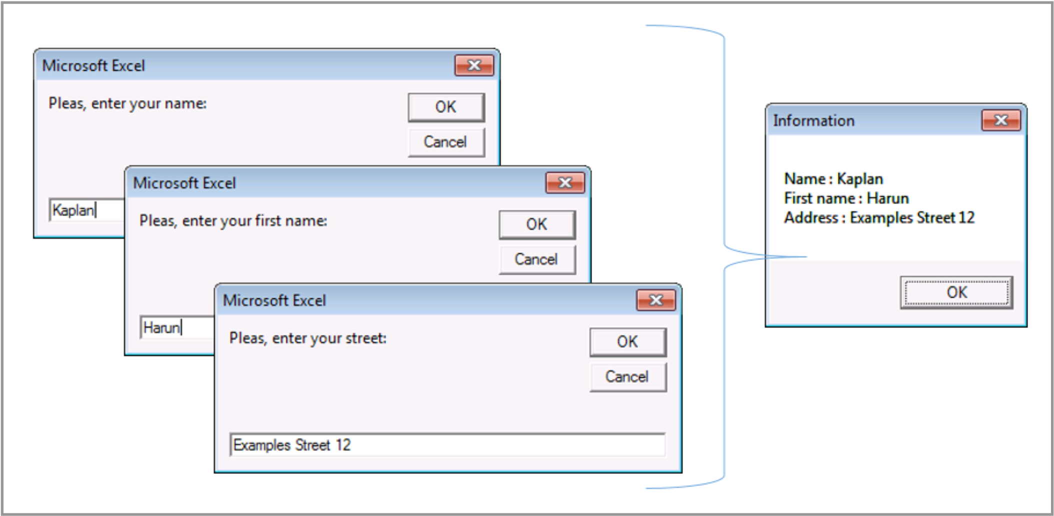


Figure 79: MsgBox with more InputBox

Download free eBooks at bookboon.com

Now the practical example from earlier with extension:

```
Sub Practical_Example_2()
repeat:
   intInput = InputBox("Please Enter your number!", _
                  Title:="Input number", _
                  Default:=5)
'Control selected button
   Select Case intInput
     Case ""
       Exit Sub
    End Select
'Control input, if input number or text
   If IsNumeric(intInput) = True Then
      If intInput < 5 Then 'by value control which number
        Title = "The value is less than 5"
        a = 2
        b = 3
        ElseIf intInput >= 5 And intInput <= 10 Then
           Title = "The value is between 5 and 10"
           a = 4
           b = 5
        ElseIf intInput >= 10 And intInput <= 15 Then
           Title = "The value is between 10 and 15"
           a = 6
           b = 7
      End If
    Else
    MsgBox ("Please enter your value!")
    GoSub repeat 'If input no number, continue with by "repeat"
    End If
   Result = a * b
   strOutput = MsgBox(("First value = " & a & vbLf & _
                   "second value = " & b & vbLf & _
                   "Result = " & Result), , Title)
End Sub
```

BIBLIOGRAPHY

Online Microsoft Developer Network:

https://msdn.microsoft.com/de-de/vba/office-vba-reference

Download free eBooks at bookboon.com