MS office offers a functionality of building complex actions and quasi-programs by means of a special scripting language called VBA (Visual Basic for Applications). In this lab, you will learn how to use the Macros module and get familiar with basic syntax and formulation of VBA in the case of Excel. Note that getting acquainted with VBA programming is central for your pair-project.

BASIC MACROINSTRUCTIONS (MACROS)

Exercise 1

MS Office offers a wide variety of built-in macros that can facilitate edition or customization of tacks in both Excel and Word. All are programmed in VBA but their code is not accessible. However, it is a good practice to test one of these to get a view on the possibilities offered for future programming. The Macros that is proposed to be tested in part of Word: the Mail Merge Manager (MMM). Note that you may find some interest in using such macro for you project. Just to make it clear: it is expected NOT TO BE USED FOR YOU OWN PAIR-PROJECT!

How to proceed. Very simple:

1. Open a blank document in Word and save it as VBA1_*yourname*.docx
2. Open a blank document in Excel and save it as VBA1_*yourname*.xlsx
3. In Excel and first spreadsheet (sheet1. You can rename it "ex1" if you wish), enter some name of people that you would like to invite, for instance, for Your birthday (e.g. column A: firstname, column B: last name). Max. 10 entries. Save and go back to word. To remember the category, write in cell A1, "firstname" and in cell A2, "lastname". You will see that this will be also very useful later on.
4. Go back to Word and write some lines of text for your birthday invitation. The purpose of MMM is to generate X instances of a document in which some fields (e.g. names of invited persons) will be automatically changed based on the Excel entries.
5. Click now on the **Mailings** tab, then select **Step by Step Mailing Merge Wizard...**
   - **a)** Choose the type of document you want to create. In our case, select **Letters**.
   - b) Click **Next: Starting document** to move to Step 2. Then, Select **Use the current document**
   - c) Click **Next: Select recipients** to move to Step 3. This step simply links with the document where are located the list of interest (in our case names in excel document). So in the new window, click on **Browse...** since your list already exists in VBA1_*yourname*.xlsx.
   - d) Select the spreadsheet where are located the data. Also, tick option **First row of data contains column header** (see previous point 3). Press OK, then in new window uncheck any list member that you don't want to use in the final merging document. Press OK.
   - e) Step 4. Place fields where they will appear in your text, e.g. firstname and lastname. For that, place the cursor in the appropriate place of you text, then click in the MMM window on **More items...** and select the fields.
   - f) After it is just a matter of preview the results and finish the merging to be ready for printing or mailing.

As already mentioned, we cannot access to the code that is behind the MMM. Fortunately, this is not always the case as it is possible in MS Office to record some macros and later on see their corresponding VBA codes. The 2 next exercises are intended to illustrate such approach.

Exercise 2

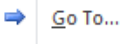
1. Enter the following data to a worksheet (e.g. in sheet2 of VBA1_*yourname*.xlsx):

location	books	papers	stamps	postcards	total
Łódź	11	35	10	10	**66**
Radomsko	13	24	12	12	**61**
Sieradz	21	8	28	28	**85**
Zgierz	8	2	16	26	**52**
total	**53**	**69**	**66**	**76**	**264**

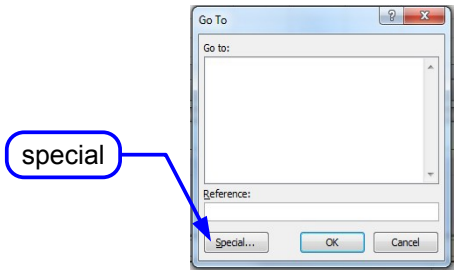
formula

Ensure, that the total values are calculated using the SUM function.

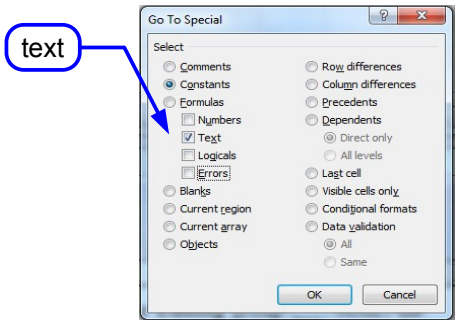
2. Formatting procedure:
   - select any single cell
   - click the **Home** tab, then click on the **Find&Select** button in the Editing group 

. Select **Go To...** 

   - in a **Go To** dialog choose **Special** button:



   - Select **constants** option and leave unchecked options **Numbers**, **Logicals** and **Errors** (only **Text** is checked):



   - click **OK**
   - format the selection as follows:

location	books	papers	stamps	postcards	total
Łódź	11	35	10	10	**66**
Radomsko	13	24	12	12	**61**
Sieradz	21	8	28	28	**85**
Zgierz	8	2	16	26	**52**
total	**53**	**69**	**66**	**76**	**264**

   - rrepeat the steps for **Formulas** resulting in **Numbers** and apply the format:

location	books	papers	stamps	postcards	total
Lodz	11	35	10	10	66
Radomsko	13	24	12	12	61
Sieradz	21	8	28	28	85
Zgierz	8	2	16	26	52
total	53	69	66	76	264

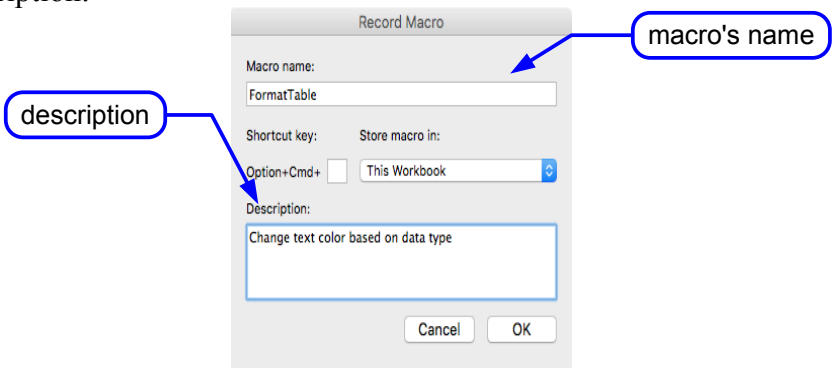
   - repeat the steps for **Constants - Numbers** and apply the format. Then, add the legend (**add new rows** and place the legend cells there):

	A	B	C	D	E	F
1		text				
2		formulas				
3		numbers				
4						
5	location	books	papers	stamps	postcards	total
6	Lodz	11	35	10	10	66
7	Radomsko	13	24	12	12	61
8	Sieradz	21	8	28	28	85
9	Zgierz	8	2	16	26	52
10	total	53	69	66	76	264

Exercise 3

Knowing, how to format the cells, you can start recording a macro.

1. Select the whole worksheet (e.g. pressing **Ctrl+A**) and then clear the formatting (click on **Clear** in the **Editing** group on **Home** tab. Select **Clear Formats** from the list). **Remove the rows** containing the legend.
2. Macro recording procedure:
   - select **Macros** on the **View** Tab → **Record Macro...**
   - call the macro with any name (but I propose to give a meaningful name, e.g. FormatTable), add the description:



   - click **OK**
   - repeat all the steps performed in the previous exercise
   - when finished, choose **Stop Recording** from the list displayed after clicking on the **Macros** button
   - in order to test the macro, again remove the worksheet format. Choose **View Macros**. Select your macro and press **Run**.
   - **Save As** your file and change the extension to .xlsm (e.g. VBA1_*yourname*.xls**m**). Why? Because your file now contains VBA macros (**m** stands for macro).

Exercise 4

Recorded macro very often needs some modifications. In order to edit the macro, you must find the worksheet containing it. In our case, we have chosen "this workbook" when creating macros, so it is enough to open macro dialog, select the macro and press **Edit** button.

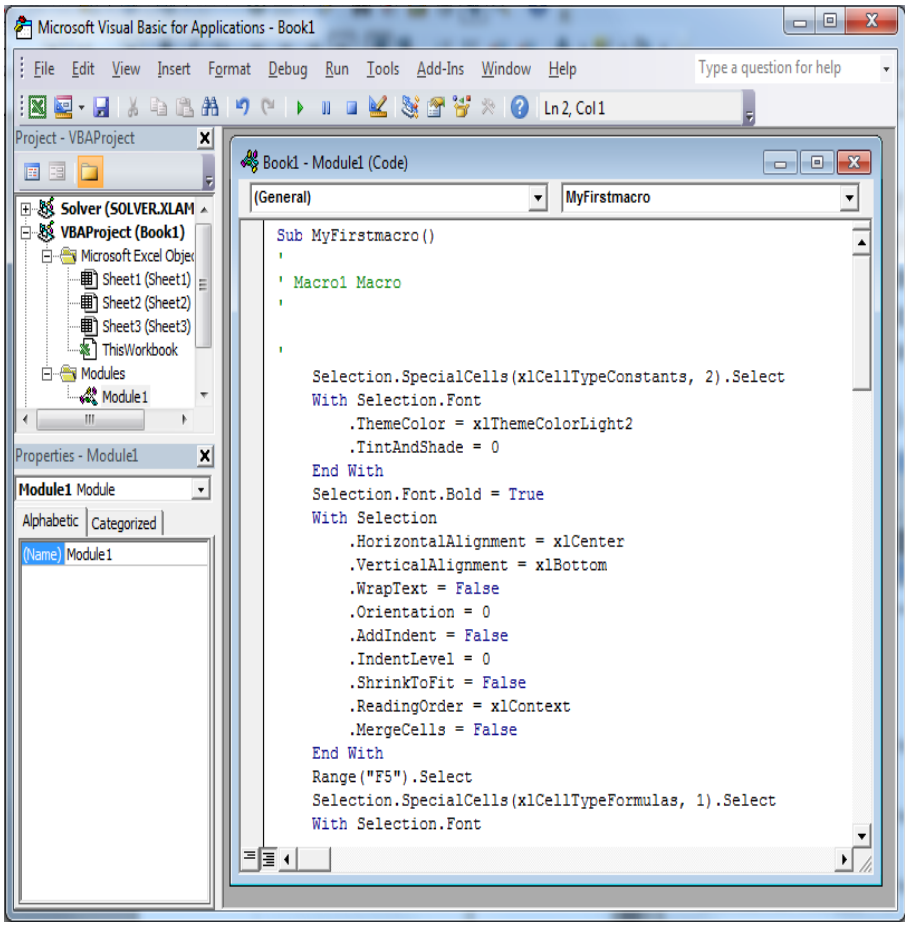
Visual Basic Editor window opens, while the standard MS Excel windows still remains open. If you want to close the editor, use **File** menu → **Close and return to Microsoft Excel** (or just close the editor).

The internal frame contains the code of our macro (note that it may look different than the example shown below). Notice, that the lines prefixed with apostrophe " ' " are not interpreted – they are the **comment lines** displayed in green. The comments (code documentation) are very important for the code conservation and maintenance. They also allow to "switch off" the code fragments that we do not want to execute, but they may be useful later. The comment explaining the code line meaning

should look as follows:

```
ColorIndex = 5 'changes colour to  blue
```

while the explanation for the whole code block should be placed as a separate line before the block.



Add a few comments:

- find the line:
  `Selection.SpecialCells(xlCellTypeConstants, 2).Select`
  add a new line before. Place there your comment (remember about the apostrophe):
  `'find and format the cells containing text`
- do the same with:
  `Selection.SpecialCells(xlCellTypeConstants, 1).Select`
  adding the comment about the number cells
- again, for:
  `Selection.SpecialCells(xlCellTypeFormulas, 1).Select`
  add the comment about cells containing formulae resulting in numbers
- find the instruction (the row number may be different):
  `Rows("1:1").Select`
  and add a comment above about code block responsible for creating the legend.

The code above shows you some characteristics of VBA programming. One main attribute is the use of so-called VBA objects that are specific for MS environments, i.e. Word, Excel of even PowerPoint. For instance, you can see above that **Range**, **Selection** or **Rows** are typical Excel objects. VBA objects are characterized by ***attributes*** (or properties, e.g. **Font**, **Orientation**) and ***methods*** (or functions, e.g. **Select**). When you change their characteristic in VBA, their appearance and content may change in the MS environment. You will learn more about objects and hierarchy of objects in the next sections.

Exercise 5

Record macro to clear format and delete the 3 rows of legend. Name it **ClearFormat**.

PROCEDURE AND FUNCTION DECLARATION

Exercise 6

Macroinstructions (macroprocedures or simply procedures) start with a keyword **`Sub`** and end with keywords **`End Sub`**. After **`Sub`** you should place the macro's name and a pair of parenthesis. Optionally, input parameters can be declared within the parenthesis. The macro body in contained between **`Sub`** and **`End Sub`**. It contains instructions executed by Visual Basic every time the macro is called.
The VBA expressions usually represent dot-expressions, where dot " . " separates an object from its field or function performed on it, e.g.:

```
Rows("1:1").Select 'select rows from 9 to 9 (in fact it is a single row)
Selection.Insert Shift:=xlDown 'insert a new row on the selected range and
                               'shift the other rows down
Range("B1").Select   'Select cell B1 to add text legend
Selection.FormulaRlC1 = "formulas" 'enter into an active cell "text" value
Range("A1").Select   'Select cell A1 to fill with corresponding color legend
With Selection.Interior 'perform the following operations on the whole
                        'selection's interior
    .ColorIndex = 3 'set color to red
    .Pattern = xlSolid 'set pattern to solid
    .PatternColorIndex = xlAutomatic 'set pattern color to automatic
End With 'end the loop
```

Note that the code above is duplicated 2 times to create legend for constant-numbers (colorindex=4, so green) and constant-text (colorindex=5, so blue). **You can easily understand here that there is a redundancy in code writing and this could be easily replaced by a new macroprocedure**. This is the aim of the following exercise instructions.

Tasks to be done:

- Declare a new procedure called CreateLegend (so write **`sub CreateLegend()`** and the **`end sub`** will appear automatically)
- Cut the code related to legend creation and paste it in the body part of the new procedure.
- Now, in order to be able to execute it, you need to call the procedure in the main procedure, i.e. in FormatTable (or the name you gave to the procedure you recorded). So, where appropriate write **`call CreateLegend.`** Execute the code (press **Run Macro** button from **Standard** tab ).

  ATTENTION: normally, you should execute first **ClearFormat** to test your procedure. However, be careful with it, because if already executed, you may delete data from the table. But, as programmers, you could later on find a fancy way to make this not possible.

As we said previously, the code is still redundant, because we use 3 times the same instruction block to create the 3 rows of legend in the Excel document. So to simplify this, we will use the possibility to pass variable inputs to the program to simplify it. Indeed, if we analyze the code, we see that **colorindex** value and the string parsed to the **FormulaR1C1** attribute (e.g. "Formulas") correspond to variables. So proceed as follows:

- Keep only one block of code
- add 2 variables between the brackets of the procedure header: one called iColorCell (declare the datatype as integer; **`colorcell As Integer`**) and the second called sLegend (declared as string).
- Now, in the procedure body, replace where needed the piece of codes by the variables (so where the cell color and the legend corresponding values have been specified).
- In FormatTable, add to the Call statement values of the variables to be parsed, e.g.:
  ```
  Call CreateLegend(3,"Formulas") 'create legend for the Formula resulting in numbers
  ```
- Click anywhere in FormatTable and run. Do you get same results?

This exercise has introduced another VBA aspect, i.e. variable declaration. You have to know that, alike C or Java, variable declaration is not obligatory (depend of case actually). If not declared, they will be automatically set as **Variant**, the VBA mastertype that takes 16 bytes for numbers and 16 bytes per character for strings. So very space consuming. So even optional, it is still a good practice to declare variables. We will see this in next exercise and in more details in lab4.

Exercise 7

This exercise and following ones show you how to create a new spreadsheet and name it, if the name hasn't been taken by another one. It also introduces a bit in more details object hierarchy, function declaration and variable declaration.

First, see below the code of the function **SheetCheck** that checks if a given worksheet exists. Copy its code to the VBA editor:

```
Function  SheetCheck(WB As Workbook,WorkSheetName As String ) As Boolean
Dim  WorkSht As Worksheet
'returns if a worksheet name is an empty string
If  Len(WorkSheetName) = 0 Then
   SheetCheck = True
Exit Function
End If
'checking if a worksheet exists
For Each  WorkSht In  WB.Sheets
   If  WorkSht.Name = WorkSheetName Then
       SheetCheck = False
       Exit Function
   End If
Next  WorkSht
SheetCheck = True
End Function
```

Try to understand the function structure. More importantly, answer the following questions:

- How many variables are parsed to the function?
- What is different between the function structure and the procedure structure?
- Which variable is returned by the function? What is so specific with VBA function declaration?

Exercise 8

Like procedures, functions are called by other functions/procedures. In our case, it will be called in a procedure that is responsible to add a new worksheet to the current Excel document. So copy the code below and again paste it to your VBA editor.

```
Sub  SimpleSheetAdd()
Dim Workbk as Workbook
Dim  NameNotExist As Boolean
Dim  BookName As String
 BookName = "SomeDocument.xlsm"
'check if a worksheet "new worksheet" exists in ".xlsm"
workbook. Replace SomeDocument by a more clever name (e.g. VBA1_yourname.xlsm)
Set  Workbk = Workbooks(BookName) 'create object
NameNotExist = SheetCheck(Workbk,"new worksheet")
'if a worksheet with name "new worksheet" does not exist, VBA creates it
If  NameNotExist Then
   Workbk.Sheets.Add
   ActiveSheet.Name = "new worksheet"
End If
Set Workbk = Nothing
End Sub
```

Few important remarks:
The previous examples show you the use of object hierarchy in Excel (but we can extend this to any MS Office software). For instance, above you can see that **Workbook** and **Worksheet** are object types, but you can also see that **Worksheet** is a sub-member of Workbook . At the top of this hierarchy is the Application object (here Excel). The "." symbol makes the link between them, e . g . `Application.Workbooks("Book1.xlsx").Worksheets("sheet1").Range("A1").Value`, which can be read as "the value of the cell "A1" in the worksheet "sheet1" of the Excel document "book1.xlsx". You can easily think of the hierarchy in MS Word also.

Again, you are not always forced to declare object variables in VBA (like standard variables) but you are highly advised to do so, first for good programming habit, and also for practical reasons: you may end with handling in one program several excel documents that will have to communicate data between each other. You can easily understand that declaring object variables may significantly reduce typing long hierarchical expressions as the one above. To do so, you will have to:

- declare object variable, usually at the beginning of the procedure/function (see examples above)
- initialize the object using the **Set** statement (see also example above).
- It is also good practice to remove from memory an object when not used using the `Set objectname= Nothing` statement (see again above).

Last but not least, the code also shows example of standard variable declarations, i.e. of type **boolean** or **string**. Even not compulsory, it is advised again to declare. Why? Because VBA is a interpreted language, it does not check by default variable declaration. This has some advantages of flexibility. However, this is prone to error (especially spelling error). If a variable contains a spelling error, it is treated as a new variable and therefore will cause syntax error. If you want to force VBA to declare all variables, you will have to write at the top of your module (before any procedure/function declaration, the `Option Explicit` statement.

Run the code above. What Happen? Any problem?

Exercise 9

Add to the code from the previous exercise some functionality so that it asks a user to enter the new worksheet name. Use:

```
NewName = InputBox("Enter a worksheet name:", "Add new worksheet", "whatever")
```

where **NewName** should be declared as:

```
Dim NewName As String
```

The prompt will appear:

Exercise 10

Create a macro prompting for a name of a worksheet to delete and then deleting it (if exists). If a worksheet does not exist, an appropriate dialog should be displayed. Use the code:

```
If Not NameNotExist Then
Workbooks(BookName).Sheets(NameToDelete).Activate
ActiveWindow.SelectedSheets.Delete
Else
MsgBox "Worksheet" & NameToDelete & " not found...", vbOKOnly, "Warning"
End If
```

The two last exercises have introduced 2 popular VBA built-in functions: M**sgbox** and I**nputbox. Msgbox** displays a message to the screen while I**nputbox** is an interrogation window waiting for an input to be parsed for a (declared or not declared) variable. Also, in the M**sgbox** first field, see the use of the ampersand "**&**" as string concatenation operator. These are popular Visual Basic functions and language symbols. A broader view of the language structure will be approached in the next lab.