



# Excel VBA Programming Golden Rules

[MasterofMacros.com](http://MasterofMacros.com)

John Franco

## **Excel VBA Programming Golden Rules**

by John Franco

© 2010 by ExcelCream.com

### **Notice of Rights**

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher.

If you want to sell this book, use it for commercial purposes, distribute it in bulk quantities in your workplace, or a hard copy version; please [Contact me](#)

### **Notice of Liability**

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and ExcelCream.com, nor its dealers or distributors, will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

### **Trademark Notice**

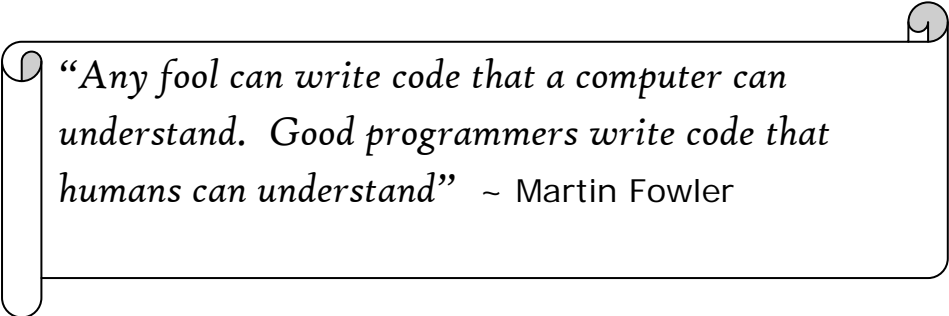
Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

## Contents

1 Introduction .....	5
2 How to make your macros easier to write, read, remember AND Maintain .....	6
2.1 Design the algorithm first.....	6
2.2 Make your Excel VBA Macro project modular.....	10
2.3 Name all your VBA elements purposefully (naming convention) .....	13
2.3.1 General guidelines for naming Subs, Functions, Modules .	13
2.3.2 General guidelines for naming Variables .....	14
2.3.3 General guidelines for naming Objects and form controls.	15
2.4 Document the macro .....	17
2.4.1 Add the main Macro information .....	18
2.4.2 Specify the purpose of the macro .....	19
2.4.3 Specify the purpose of any significant line or block of code .....	20
2.5 Make the code flow.....	20
2.5.1 Indenting.....	20
2.5.2 Line breaks .....	22
3 How to make macros/Functions/statements dynamic .....	23
3.1 Put changing-information outside .....	24
3.1.1 Placeholders in Subs & Functions.....	24
3.1.2 Placeholders in VBA statements and expressions .....	26
4 How to write macros optimized for speed and memory .....	27
4.1 Screen update .....	27

4.2 Selections .....	27
4.3 Object variables use .....	28
4.4 Excel recalculation.....	29
4.5 Not declaring Variables .....	30

## 1 INTRODUCTION



*“Any fool can write code that a computer can understand. Good programmers write code that humans can understand” ~ Martin Fowler*

This VBA best practice manual was written from a non-developer point of view, I am a professional who uses Excel as a tool and not as an end.

In other words, I don't earn money by developing macros but by becoming more efficient through them.

I did the best I can for compiling and summarizing the best practices that I have used in my professional life, there are more details about each technique and you can explore more if you want. **There are specialized books and blogs out there that can take you deep in this area.**

I guarantee you can enhance your programming proficiency and results significantly by implementing these techniques.

I hope you **enjoy the “Excel VBA Programming Golden Rules”** manual and also put into practice the VBA programming tips and techniques I share with you.

To your Excel VBA Macro success!

John Franco

<http://www.masterofmacros.com>

## 2 HOW TO MAKE YOUR MACROS EASIER TO WRITE, READ, REMEMBER AND MAINTAIN

A simple macro is good for both ones:

- Good for a computer because it requires less processing time and resources
- Good for users because they can understand code easily and intuitively and also because they will enjoy faster results derived from faster macros

Here are the main guidelines to make your macros friendlier...

### 2.1 Design the algorithm first

Yes, the best way to keep your VBA macro code friendly is to make it simple.

How do you build simple macros?

By removing redundancies and inefficiencies from the logic!

How do you make your Excel Macros that elegant?

**You already know that the best way for developing strong macro logics is by planning and diagraming your Excel VBA macro before you write it.**

*"Flowcharting/planning is a low risk activity because you remove/rearrange/change thoughts not real things. When you are planning your house, you reorganize walls, pillars, schedule and budget on your blueprint instead of relocating the physical items on site. When you are planning your macro, you rearrange logic on your flowchart instead of debugging or rewriting bunches of VBA code. At the end you will always spend less time and resources solving/anticipating your project problems on a paper scheme than on real life." – 7 secrets of professional and effective flowcharts*

And also...

No matter the amount of improvements you add to ship transportation technology, they will never become airplanes.

The same with macros...

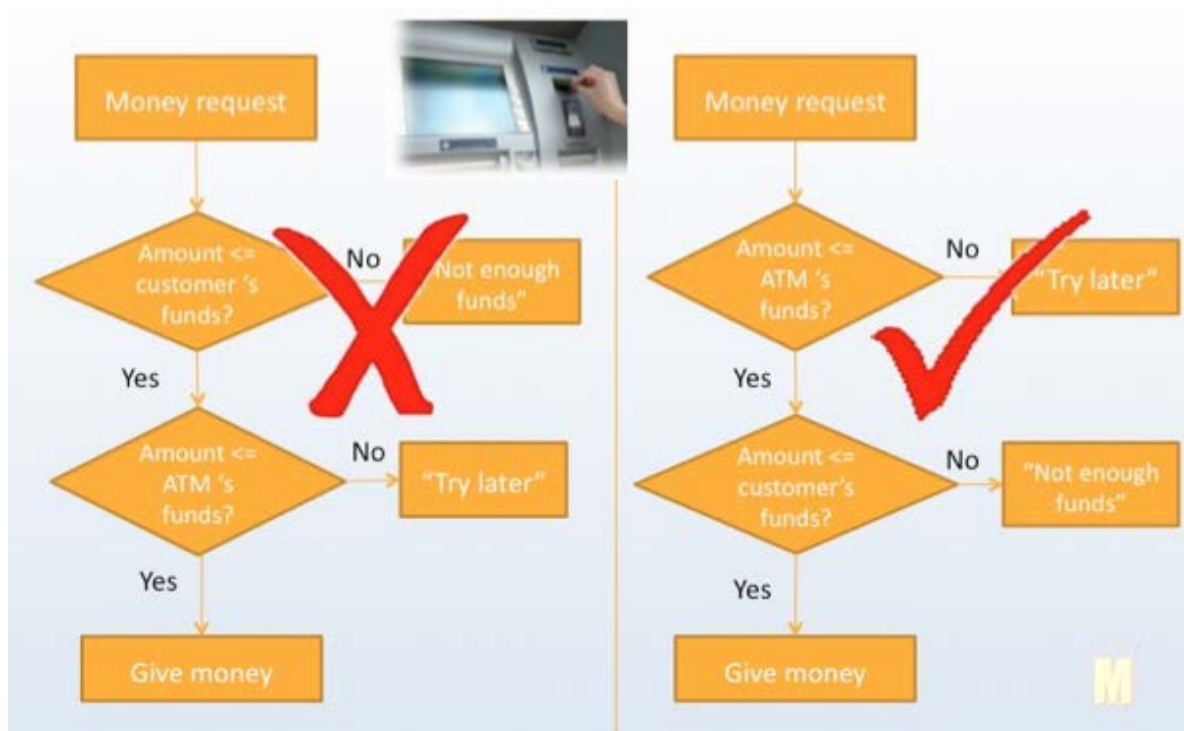
You can change a line of code of a poor macro; but by doing that you are improving a broken model, no matter how many improvements you add to the poor macro, it will keep a poor macro.

And dealing with poor macros burns you and your PC because you both will need to double the processing time and resources (writing/understanding/debugging/redoin the macro).

Here's a simple example of a redundant macro...

Refer to the picture below.

The solution at the right checks customer's funds availability only after it has checked that the ATM machine has money. Why? Because it does not make sense to check customer's funds first to later say to him, sorry this ATM machine has no money.



The point is...

You cannot improve a macro by just tweaking a line of code here and there; see the ATM example below...

For these variables...

CustomerAmount = \$100

CustomerFunds = \$500

ATMFunds = \$50

A right algorithm should return "Try later, this ATM machine has no money" or something like that.

The macro at the right retrieves that kind of message after performing two checks (customer funds and ATM funds)...

You can improve that macro by making each line of code better, for example...

You can add a Title to the message, see below...

```
(General)

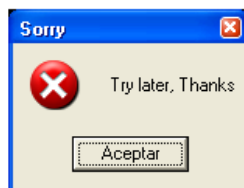
Public Sub RequestMoney_1()
    CustomerAmount = 100
    CustomerFunds = 500
    ATMFunds = 50

    If CustomerAmount >= CustomerFunds Then
        MsgBox "Not enough funds", vbCritical
    ElseIf CustomerAmount >= ATMFunds Then
        MsgBox "Try later", vbCritical
    Else
        MsgBox "Take your money", vbCritical
    End If
End Sub
```



```
Public Sub RequestMoney_3()
    CustomerAmount = 100
    CustomerFunds = 500
    ATMFunds = 50

    If CustomerAmount >= CustomerFunds Then
        MsgBox "Not enough funds", vbCritical, "Sorry"
    ElseIf CustomerAmount >= ATMFunds Then
        MsgBox "Try later, Thanks", vbCritical, "Sorry"
    Else
        MsgBox "Take your money", vbCritical
    End If
End Sub
```



But the macro will remain poor; it will perform two checks instead of only one.

So, to make a macro flow so requires less code and is easily understood, you need to make disruptive changes (not incremental ones), and this is done at the logic level.

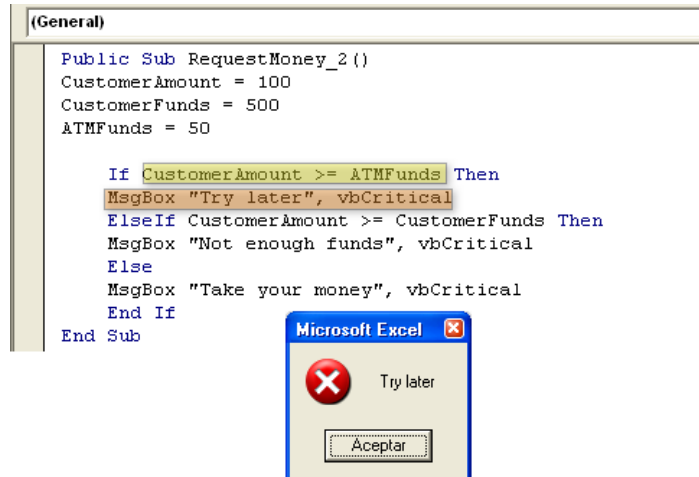


See the macro at the right, it returns the same message but after performing one check (ATM funds first)

The logic level (algorithm) can be efficiently improved when you see how the parts connect.

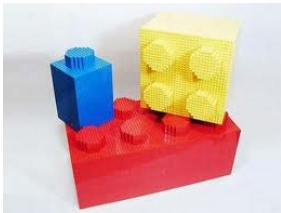
And flowcharting

allows you to see the connections between elements so you remove, rearrange the code at the structure level, instead at the line of code level.



### 2.2 Make your Excel VBA Macro project modular

How do you build a Lego castle, ship, city, etc?



Using 3 basic types of blocks, right?

So you can build systems through units.

You can shape a Bart Simpson (see picture at the right) or you can build a nice Dr. Elephant and his patience (see picture below).



Here's a Dictionary definition for "module"...

*"Module - each of a set of standardized parts or independent units that can be used to construct a more complex structure, such as an item of furniture or a building"*

Some examples of real world modules are:

- Bricks
- Columns
- Walls
- Beams
- Floors

Using modules is not only good to build bigger structures.

You can also...

- Compose different shapes than those of the modules, for example you can build a wall using bricks, or you can build a cube using bricks too

- One bad news with real world modules is that improving a unit at the fabric doesn't replace the already fixed units in previously constructed buildings; on the other hand, when you improve a VBA procedure, all the Macros where the procedure is used, take advantage of the improvements.

```
(General) Macro3  
  
Public Sub OpenWorkbookDialog()  
'developed by John Franco  
'MasterofMacros.com  
  
'separate the wildcard expressions with semicolons; for example, "Visual  
'If FileFilter is omitted, this argument defaults to "All Files (*.*)",*.  
  
FileToOpen = Application.GetOpenFilename("Excel Files (*.xl*), " & "*.xl*",  
  
    'If Cancel then exit  
    If FileToOpen = False Then  
        Exit Sub  
    End If  
  
'Open selected file  
Workbooks.Open FileToOpen  
  
End Sub  
  
Public Sub Macro1()  
'>>>>>>>some code here<<<<<<<<<<  
OpenWorkbookDialog  
'>>>>>>>some code here<<<<<<<<<<  
End Sub  
  
Public Sub Macro2()  
'>>>>>>>some code here<<<<<<<<<<  
OpenWorkbookDialog  
'>>>>>>>some code here<<<<<<<<<<  
End Sub  
  
Public Sub Macro3()  
'>>>>>>>some code here<<<<<<<<<<  
OpenWorkbookDialog  
'>>>>>>>some code here<<<<<<<<<<  
End Sub
```

When you develop your macros as units, for example: importing file, processing data, exporting file, etc. you can work on every single unit and affect the whole system (every instance of each unit).

How do you make a VBA macro project modular?

1. Put specific outcomes into Function procedures: commission calculation, convert temperature from Celsius to Fahrenheit, etc.
2. Put specific processes into Sub procedures: Opening a workbook, formatting a report, inserting a chart, etc.
3. Group common Sub/Function procedures into Modules: directories and folders procedures, charting procedures, sales procedures, etc.

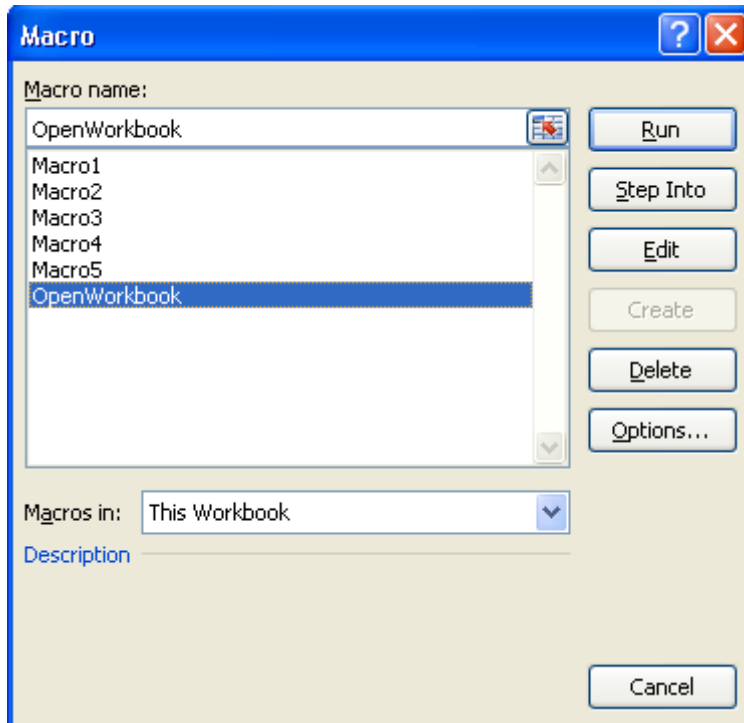
### 2.3 Name all your VBA elements purposefully (naming convention)

You create Macros for reusing them, right?

So a good name increases the usability of a Macro. It allows you to pick the right Macro on the dialog or on the code window.

For example:  
compare these names "Macro1", "OpenWorkbook".  
See picture at the right...

You can discern what the Macro does by looking at the name.  
Additionally, you will remember Macros weeks or months later.



A good practice is to start a Macro name with a verb followed by the name of the object it affects (start each section in uppercase), for example:

- OpenWorkbook
- CloseExcel
- DeleteFormat
- ChangeColorCell
- ChangeColorFont

Learn more details about naming your VBA elements...

#### 2.3.1 General guidelines for naming Subs, Functions, Modules

- Begin each separate word in a name with a capital letter, as in FindLastRecord and RedrawMyForm
- Begin function and method names with a verb, as in InitNameArray or CloseDialog and include the name of the object the method/function modifies, for example: CloseWorkbook, DeletePivotTable, etc.

- Begin class and property names with a noun, as in EmployeeName or CarAccessory, SheetName
- Use abbreviations to keep name lengths reasonable, for example, "HTML", instead of "Hypertext Markup Language". Keep the abbreviations consistent between projects so you avoid confusions

### 2.3.2 General guidelines for naming Variables

- Begin each separate word in a name with a capital letter, as in FindLastRecord and RedrawMyForm
- Avoid using names in an inner scope that are the same as names in an outer scope. Errors will result if the wrong variable is accessed. If a conflict occurs between a variable and the keyword of the same name, you must identify the keyword by preceding it with the appropriate scope prefix. See table below

Scope	Prefix	Example
Module-level	m	mstrCompName
Local to procedure	None	intNum

- Use a prefix to differentiate the data type the variable contains (integer, Double, etc.), for example: intMonth, intAge, dblSales, etc. This way you will know what data you can store on any given variable and avoid "Unexpected data type" errors. See the most common data type prefixes below...

Data type	Prefix	Example
Boolean	bln	blnFound
Currency	cur	curRevenue
Date (Time)	dtm	dtmStart
Double	dbl	dblTolerance
Integer	int	intQuantity

Long	lng	lngDistance
Single	sng	sngAverage
String	str	strFName
Variant	vnt	vntCheckSum

- Force VBA to declare variables by using the VBA statement "Option Explicit", this way you avoid typo errors when typing your variables. For example, you might accidentally type "intYear = 23" instead of "intYears = 23". Avoid assigning values to different variables (your variable intYears does not have any value assigned because you stored a value in the variable intYear). Use "Option Explicit" VBA statement to force you to declare all variables before using them, this error would be highlighted by the VBA compiler, as the variable "intYear" would not have been declared

### 2.3.3 General guidelines for naming Objects and form controls

Name forms and controls using the prefixes in the table below...

Control type	prefix	Example
Check box	chk	chkReadOnly
Combo box, drop-down list box	cbo	cboEnglish
Command button	cmd	cmdExit
Form	frm	frmEntry
Frame	fra	fraLanguage
Label	lbl	lblHelpMessage
Line	lin	linVertical
List box	lst	lstPolicyCodes
Option button	opt	optGender
Picture box	pic	picVGA

Shape	shp	shpOctagon1
Text box	txt	txtLastName



### 2.4 Document the macro

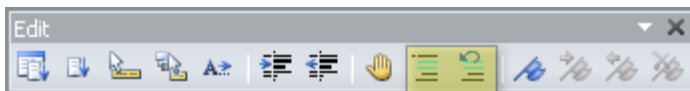
*Good code is its own best documentation. As you're about to add a comment, ask yourself, "How can I improve the code so that this comment isn't needed?"*  
~ Steve McConnell

You did a great documenting work by naming VBA elements properly and by planning the logic of the macro, but here is another piece of the documenting process you should always have right.

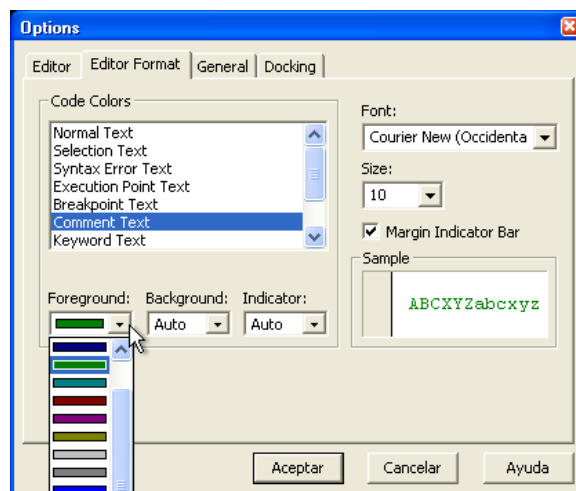
Macros are moved, copied, edited, etc. so the best way to document a macro is by putting all the relevant information inside the macro. This way the documentation is always accompanying the macro and is visible to the macro developer or to those who will run the macro in the future.

So you need to comment your code.

You insert a comment by using a leading apostrophe (') on any line of code or by using the edit toolbar...

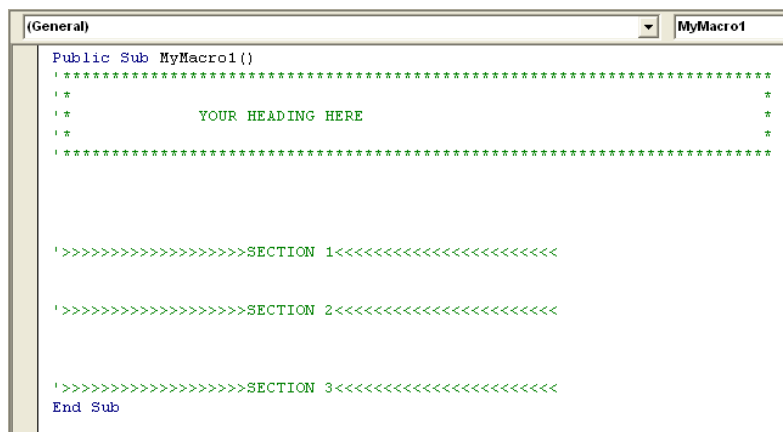
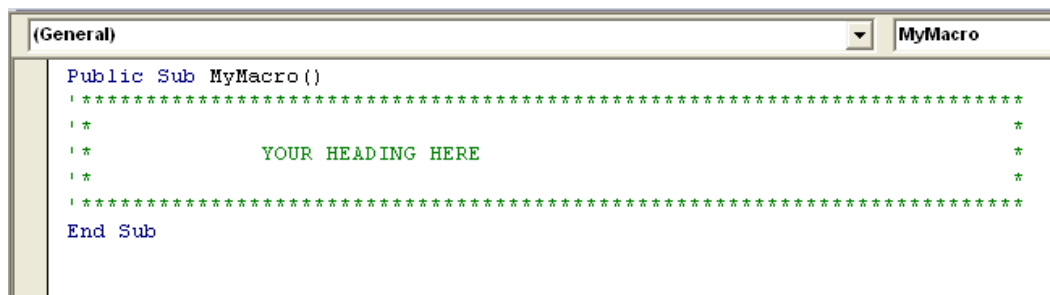


Comments appear green by default but they can be changed, go to  
Tools>Options>Editor  
Format>Comment Text...



1000

- Last update {Date here}
- Version N {macro versions here}
- Changes



### 2.4.2 Specify the purpose of the macro

You always need to know what to expect from a macro before you run it. For example: where to put the cursor, what sheet to activate, etc.

This information is very important when you are accessing macros that were written by others and also when you are accessing your own macros that were written some time ago.

Here's what you need to include as the full specs of a macro...

#### *Purpose*

What the macro does (not how it does it)

#### *Assumptions*

External variable, control, file, etc. accessed by the macro

#### *Effects*

Each affected external variable, control, or file, and the effect it has (only if it is not obvious).

#### *Inputs*

What the macro needs to start computations and produces the outputs (returns)?

- A file name
- Values in specific cells
- A Yes/No user input using an input box
- A number
- A date
- What arguments: number of years, amount of sheets to print, etc.
- Etc.

#### *Returns*

What your macro returns?

- An integer
- A new sheet

- A new pivot table
- A formula
- A new value in a given cell
- Etc.

### 2.4.3 Specify the purpose of any significant line or block of code

Do this when the purpose of any given line/block of code is not so obvious.

Here's an example...

```
'add row fields
With ActiveSheet.PivotTables("SalesSummary").PivotFields("Sales Person")
    .Orientation = xlRowField
    .Position = 1
End With

'add data fields
'Jan-08
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").PivotFields
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Jan-08").NumberFormat = "#,##0.00"
'Feb-08
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").PivotFields
```

## 2.5 Make the code flow

### 2.5.1 Indenting

Indenting is adding spaces or tabs to a line of code so it's moved at some point to the right. Each indenting distance can be considered as a level. See below...

No indenting

First indenting level

Second indenting level

Third indenting level

Indenting has no effect on how your VBA macro code runs.

Why should you use indenting?

Loops (for next, do while, do until) and condition structures (if then else, select case) have a start and an end and they usually go nested, so it's a good practice to indent each block at the appropriate level so you...

- Know how the code flows. When you have nested the same kind of loop or condition structure multiple times (such as one "For

Next" inside another one), you don't need to figure out which "end of loop" goes with which "start of loop"

- Get oriented in the jungle of code quickly because you can see at a glance where the end of a code block is (eg: where is the End If/End Select/Next for this If/Select Case/For?), rather than having to read each line until you find it
- Focus on the code you are working on by immediately ignoring chunks of irrelevant code
- Detect missing/messed line of codes easily (eg: a missing "End If" for a condition structure)

See example below...

Block 1 start

    Block 2 start

        Block 3 start

        Block 3 end

    Block 2 end

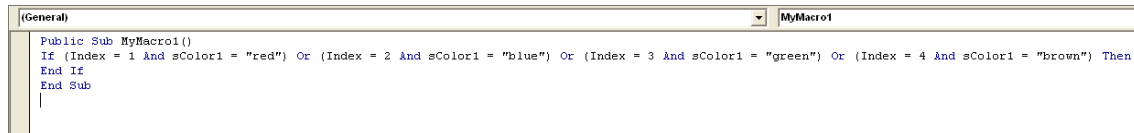
Block 1 end

### 2.5.2 Line breaks

Your code can also be made to flow by inserting line breaks in the middle of long lines of code.

Just add a space followed by an underscore ( `_` ) just before the line break. This tells the VBA compiler that the current line of code continues on the following line.

This is a long line of code...



By adding line breaks the code can be presented as follows:

```
Public Sub MyMacro2()  
If (Index = 1 And sColor1 = "red") Or _  
    (Index = 2 And sColor1 = "blue") Or _  
    (Index = 3 And sColor1 = "green") Or _  
    (Index = 4 And sColor1 = "brown") Then  
End If  
End Sub
```

### 3 HOW TO MAKE MACROS/FUNCTIONS/STATEMENTS DYNAMIC

Imagine you work on Wal-Mart, the biggest retailer in the world, and your main duty of the current week is labeling (\$1.45 price label) all the tomato sauces bottles (let's say they are 1000 bottles); after two days of boring work, you finish your uninteresting tagging task.

But bad news...

As soon as you are done, your boss comes to you and order to change the price tags from \$1.45 to \$1.46, so you need to get back to the shelves and replace each price tag by the new one. This tagging approach is a static solution.

Now imagine this happy situation...

Your duty is the same one, labeling 1000 bottles but now you put barcode labels to each bottle.

So customers can check the price on a barcode machine near the tomato sauces shelf.



Now, the price is not on the bottle but on some central database. You add barcode labels on bottles 1000 times; the same number of times as the price tags, but here's the great news...

You can update the price of 1000 bottles by making just one change at the central database.

Now you have fun!

So when your boss asks you to change the product price, you just change the respective database entry from \$1.45 to \$1.46 and all

your bottles will be updated, yes at once! Now, each time a customer checks the barcode on the machine, he will get the latest price (\$1.46). This tagging approach is a dynamic solution.

So when you make a Sub (macro), Function or VBA statement dynamic what you are doing is putting all the changing-information outside them.

Let's say this again...

### **Make a structure dynamic by putting the changing-information outside it.**

A dynamic structure works for different values without changing the structure of the code.

You can make a Sub (macro), Function or VBA statement dynamic by using a very simple technique...

#### **3.1 Put changing-information outside**

The formula  $=2+2$  will always be 4, on the other hand the formula  $=A1+A2$  will have different results depending on the A1 and A2 values. See picture at the right

	A
1	2
2	3
3	

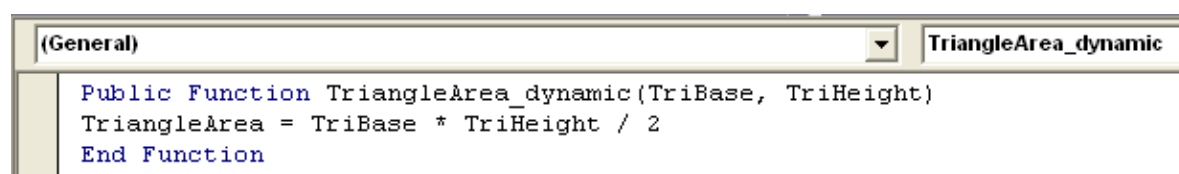
So the secret to build this dynamic formula was putting placeholders that refer to outside values (changing-information is outside), so you put A1 and A2 instead of hard-coded values.

Let's explore these techniques in more detail now...

#### **3.1.1 Placeholders in Subs & Functions**

You can use placeholders in VBA macros and functions by using arguments.

This is a dynamic function...



As you can see the changing-information (Triangle base and Triangle height) is outside the function. So you can call this function using different values, see below...



```
Public Sub Test()  
GiveMeTheAreaOfTriangle = TriangleArea(33, 4)  
End Sub
```

```
Public Sub Test()  
GiveMeTheAreaOfTriangle = TriangleArea(10, 15)  
End Sub
```

Here's another example...

This macro prints always 1 copy of sheets 1 to 10.

```
Public Sub PrintSheets()  
'developed by John Franco  
'MasterofMacros.com  
  
ActiveSheet.PrintOut from:=1, To:=10, Copies:=1  
End Sub
```

On the other hand, this macro prints any range of pages and any amount of copies each time...

```
Public Sub PrintSheets_dynamic(FromPage, ToPage, CopiesAmount)  
'developed by John Franco  
'MasterofMacros.com  
  
ActiveSheet.PrintOut from:=FromPage, To:=ToPage, Copies:=CopiesAmount  
End Sub
```

### 3.1.2 Placeholders in VBA statements and expressions

And you can use placeholders in VBA statements by using variables. Here is a simple example...

```
Public Sub PrintSheets_dynamic2()  
    'developed by John Franco  
    'MasterofMacros.com  
  
    FromPage = 1  
    ToPage = 10  
    CopiesAmount = 22  
  
    ActiveSheet.PrintOut from:=FromPage, To:=ToPage, Copies:=CopiesAmount  
End Sub
```

You can do this with any VBA statement or expression:

- InputBox(prompt[, title] [, default] [, xpos] [, ypos] [, helpfile, context])
- MsgBox(prompt[, buttons] [, title] [, helpfile, context])
- Etc.

## 4 HOW TO WRITE MACROS OPTIMIZED FOR SPEED AND MEMORY

Here you will learn what consumes more time and resources in macros and how to optimize these issues...

### 4.1 Screen update

Each time you perform one action like typing a text, inserting a chart, etc. your computer screen is updated, you don't notice this since you are doing these activities separated in time; but when you run macros, the computer does all those tasks in a couple of minutes, so this time is worth to turn the screen updating off.

Doing this is like jumping to the desired movie scene instead of watching a chain of quick-motion sequences before you watch it.

When you do this, your macro will not spend computing resources to update the screen at every action it performs.

- Shifting sheets
- Range selections
- Chart creation
- Table creation
- And other outputs

How do you do that?

It's very simple...

Write this instruction at the beginning of your macro (after the "Public Sub ()" line)...

`Application.ScreenUpdating = False` (without quotation marks)

And this one (optional) at the end of your macro (before the "End Sub" line)...

`Application.ScreenUpdating = True` (without quotation marks)

### 4.2 Selections

You don't need to select cells/sheets to work with them so avoid doing this.

So to change the content of some cells you don't need to do this...

```
Range("A1").Select
```

```
Range("A1").Value = "Hello World"
```

```
Range("A2").Select
```

```
Range("A2").Value = "Hello World"
```

Just do this...

```
Range("A1").Value = "Hello World"
```

```
Range("A2").Value = "Hello World"
```

And the same when you change properties of a sheet for example...

```
Sheets("Sheet1").Select
```

```
Sheets("Sheet1").Tab.Color= 1
```

Just do this...

```
Sheets("Sheet1").Tab.Color= 1
```

Here's another example...

You don't need to select cells in other sheets to work with them, see below...

```
Workbooks("Book2").Sheets("Sheet1").Range("A1").Select
```

```
Range("A1").Value = "Hello World"
```

Just do this...

```
Workbooks("Book2").Sheets("Sheet1").Range("A1").Value="Hello World"
```

Or even better...

If you are going to refer to a VBA object again and again, store it into an object variable.

### **4.3 Object variables use**

Instead of using this line of code over and over

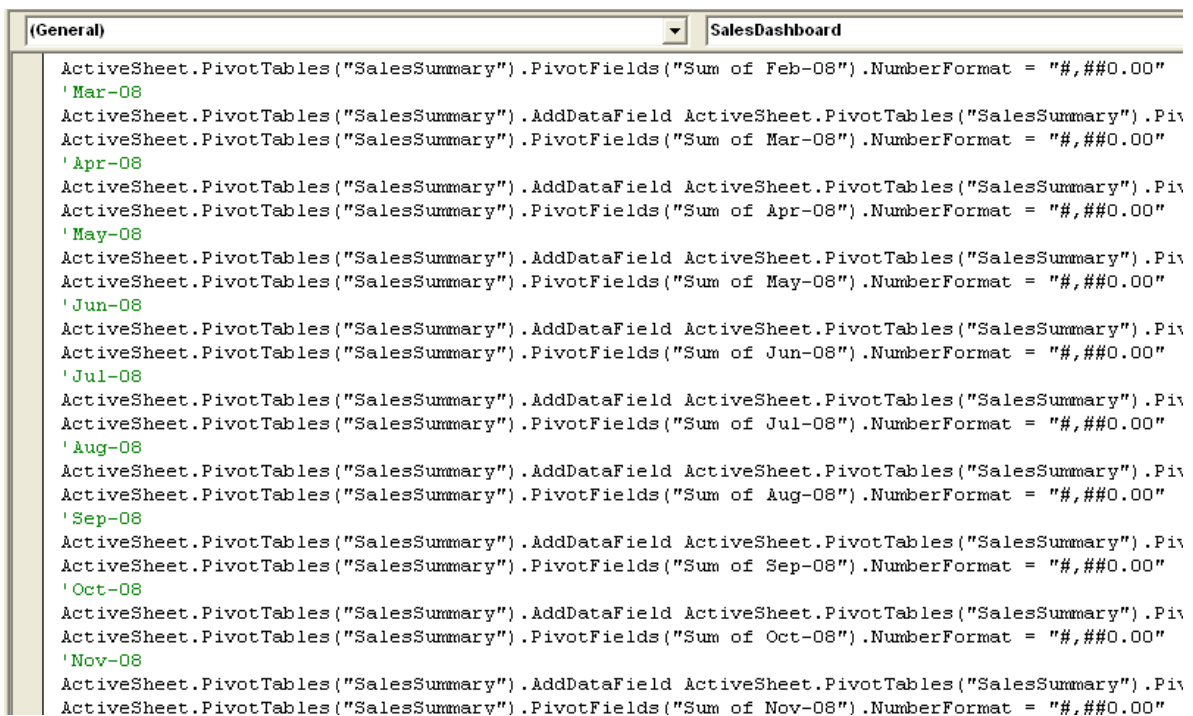
```
Workbooks("Book2").Sheets("Sheet1").Range("A1").Value="Hello World"
```

Store the cell A1 on an object variable this way...

```
Sub Macro2()  
Dim MyRange As Range  
  
Set MyRange = Workbooks("Book2").Sheets("Sheet1").Range("A1")  
  
MyRange = "Hello World"  
  
End Sub
```

Here's an example of a VBA object used over and over again...

The line `ActiveSheet.PivotTables("SalesSummary")` will be used again and again in the macro. See below...



```
(General) SalesDashboard  
  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Feb-08").NumberFormat = "###0.00"  
'Mar-08  
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").Piv  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Mar-08").NumberFormat = "###0.00"  
'Apr-08  
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").Piv  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Apr-08").NumberFormat = "###0.00"  
'May-08  
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").Piv  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of May-08").NumberFormat = "###0.00"  
'Jun-08  
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").Piv  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Jun-08").NumberFormat = "###0.00"  
'Jul-08  
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").Piv  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Jul-08").NumberFormat = "###0.00"  
'Aug-08  
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").Piv  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Aug-08").NumberFormat = "###0.00"  
'Sep-08  
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").Piv  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Sep-08").NumberFormat = "###0.00"  
'Oct-08  
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").Piv  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Oct-08").NumberFormat = "###0.00"  
'Nov-08  
ActiveSheet.PivotTables("SalesSummary").AddDataField ActiveSheet.PivotTables("SalesSummary").Piv  
ActiveSheet.PivotTables("SalesSummary").PivotFields("Sum of Nov-08").NumberFormat = "###0.00"
```

### 4.4 Excel recalculation

As your macro alters the value of sheets, it's inevitable that linked formulas are updated. When you have big spreadsheets this will take your performance down.

In case you are not using those results/outputs yet, turn Excel calculation off and turn it on when you require it.

Use this simple VBA instruction to turn calculation off...

```
Application.Calculation = xlManual
```

And this one to turn calculation on again...

```
Application.Calculation = xlAutomatic
```

## 4.5 Not declaring Variables

You can if you wish not declare a variable and still use it to store a Value (text, number, Boolean, etc.) or an Object (Range, Workbook, Chart, Word application, etc.). Unfortunately this comes at a price. The price is slower running of macro code.

If you are using variables which have not been dimensioned, Excel (by default) will store them as the Variant data type. This means that Excel will need to decide each time a variable is assigned a value what data type it should be.

See the table below to become aware of how much memory each variable type requires...

Data type	Storage size	Range
Byte	1 byte	0 to 255
Boolean	2 bytes	True or False
Integer	2 bytes	-32,768 to 32,767
Long (long integer)	4 bytes	-2,147,483,648 to 2,147,483,647
LongLong (LongLong integer)	8 bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 (Valid on 64-bit platforms only.)
LongPtr (Long integer on 32-bit systems, LongLong integer on 64-bit systems)	4 bytes on 32-bit systems, 8 bytes on 64-bit systems	-2,147,483,648 to 2,147,483,647 on 32-bit systems, -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 on 64-bit systems
Single (single-precision floating-point)	4 bytes	-3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values
Double (double-precision floating-point)	8 bytes	-1.79769313486231E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values
Currency (scaled integer)	8 bytes	-922,337,203,685,477.5808 to 922,337,203,685,477.5807
Decimal	14 bytes	+/-79,228,162,514,264,337,593,543,950,335 with no decimal point; +/- 7.9228162514264337593543950335 with 28 places to the right of the decimal; smallest non-zero number is +/- 0.00000000000000000000000001
Date	8 bytes	January 1, 100 to December 31, 9999
Object	4 bytes	Any Object reference
String (variable-length)	10 bytes + string length	0 to approximately 2 billion
String (fixed-length)	Length of string	1 to approximately 65,400
Variant (with numbers)	16 bytes	Any numeric value up to the range of a Double

Variant (with characters)	22 bytes + string length (24 bytes on 64-bit systems)	Same range as for variable-length String
User-defined (using Type)	Number required by elements	The range of each element is the same as the range of its data type.

How you declare variables?

Use the Dim statement. See example below...

```
Sub Macro3()  
    Dim MyRange As Range  
    Dim MyString As String  
    Dim MyAge As Integer  
  
End Sub
```