Getting Started with VBA in Excel 2010

Office 2010

Summary: Introduces Visual Basic for Applications (VBA) in Excel 2010 to the Excel power user who is not yet a programmer. This article includes an overview of the VBA language, instructions on how to access VBA in Excel 2010, a detailed explanation of a solution to a real-world Excel VBA programming problem, and tips on programming and debugging. (14 printed pages)

Ben Chinowsky, SDK Bridge

November 2009

Applies to: Microsoft Excel 2010

Contents

- Why Use VBA in Excel 2010?
- VBA Programming 101
- Macros and the Visual Basic Editor
- A Real-World Example
- Modifying the Recorded Code
- More Things that You Can Do with VBA
- What's Next?
- Additional Resources

Why Use VBA in Excel 2010?

Microsoft Excel 2010 is an extremely powerful tool that you can use to manipulate, analyze, and present data. Sometimes though, despite the rich set of features in the standard Excel user interface (UI), you might want to find an easier way to perform a mundane, repetitive task, or to perform some task that the UI does not seem to address. Fortunately, Office applications like Excel have Visual Basic for Applications (VBA), a programming language that gives you the ability to extend those applications.

VBA works by running macros, step-by-step procedures written in Visual Basic. Learning to program might seem intimidating, but with some patience and some examples such as the ones in this article, many users find that learning even a small amount of VBA code makes their work easier and gives them the ability to do things in Office that they did not think were possible. Once you have learned some VBA, it becomes much easier to learn a whole lot more—so the possibilities here are limitless.

By far the most common reason to use VBA in Excel is to automate repetitive tasks. For example, suppose that you have a few dozen workbooks, each of which has a few dozen worksheets, and each of those needs to have some changes made to it. The changes could be as simple as applying new formatting to some fixed range of cells or as complex as looking at some statistical characteristics of the data on each sheet, choosing the best type of chart to display data with those characteristics, and then creating and formatting the chart accordingly.

Either way, you would probably rather not have to perform those tasks manually, at least not more than a few times. Instead, you could automate the tasks by using VBA to write explicit instructions for Excel to follow.

VBA is not just for repetitive tasks though. You can also use VBA to build new capabilities into Excel (for example, you could develop new algorithms to analyze your data, then use the charting capabilities in Excel to display the results), and to perform tasks that integrate Excel with other Office applications such as Microsoft Access 2010. In fact, of all the Office applications, Excel is the one most used as something that resembles a general development platform. In addition to all the obvious tasks that involve lists and accounting, developers use Excel in a range of tasks from data visualization to software prototyping.

Despite all of the good reasons to use VBA in Excel 2010, it is important to remember that the best solution to a problem might not involve VBA at all. Excel has a large range of features even without VBA, so even a power user is unlikely to be familiar with them all. Before you settle on a VBA solution, search the Help and online resources thoroughly to make sure that there is not a simpler way.
VBA Programming 101

Using Code to Make Applications Do Things

You might think that writing code is mysterious or difficult, but the basic principles use every-day reasoning and are quite accessible. The Office 2010 applications are created in such a way that they expose things called objects that can receive instructions. You interact with applications by sending instructions to various objects in the application. These objects are many, varied, and flexible, but they have their limits. They can only do what they are designed to do, and they will only do what you instruct them to do.

Objects

Programming objects relate to each other systematically in a hierarchy called the object model of the application. The object model roughly mirrors what you see in the user interface; for example, the Excel object model contains the Application, Workbook, Sheet, and Chart objects, among many others. The object model is a conceptual map of the application and its capabilities.

Properties and Methods

You can manipulate objects by setting their Properties and calling their Methods. Setting a property changes some quality of the object. Calling a method causes the object to perform some action. For example, the Workbook object has a Close method that closes the workbook, and an ActiveSheet property that represents the sheet that is currently active in the workbook.

Collections

Many objects come in both singular and plural versions—Workbook and Workbooks, Worksheet and Worksheets, and so on. The plural versions are called collections. Collection objects are used to perform an action on multiple items in the collection. Later on, this article discusses how to use the Worksheets collection to change the name of each worksheet in a workbook.

Macros and the Visual Basic Editor

Now that you know something about how Microsoft Excel 2010 exposes its object model, you can try calling object methods and setting object properties. To do so, you must write your code in a place and in a way that Office can understand; typically, by using the Visual Basic Editor. Although it is installed by default, many users do not know that it is even available until it is enabled on the ribbon.

Developer Tab

All Office 2010 applications use the ribbon. One tab on the ribbon is the Developer tab, where you access the Visual Basic Editor and other developer tools. Because Office 2010 does not display the Developer tab by default, you must enable it using the following procedure:

To enable the Developer tab

1. On the File tab, choose Options to open the Excel Options dialog box.
2. Click Customize Ribbon on the left side of the dialog box.
3. Under Choose commands from on the left side of the dialog box, select Popular Commands.
4. Under Customize the ribbon on the right side of the dialog box, select Main tabs, and then select the Developer check box.
5. Click OK.

After Excel displays the Developer tab, note the location of the Visual Basic, Macros, and Macro Security buttons on the tab.

Figure 1. Developer tab in Excel 2010
Security Issues

Click the Macro Security button to specify which macros can run and under what conditions. Although rogue macro code can seriously damage your computer, security conditions that prevent you from running helpful macros can seriously undermine your productivity. Macro security is a complex and involved topic that you should study and understand if you work with Excel macros.

For the purposes of this article, be aware that if the Security Warning: Macros have been disabled bar appears between the ribbon and the worksheet when you open a workbook that contains a macro, you can click the Enable Content button to enable the macros.

Also, as a security measure, you cannot save a macro in the default Excel file format (.xlsx); instead, you must save the macro in a file with a special extension, .xlsm.

Visual Basic Editor

This following procedure shows you how to create a new blank workbook in which to store your macros. You can then save the workbook in the .xlsm format.

To create a new blank workbook

1. Click the Macros button on the Developer tab.
2. In the Macro dialog box that appears, type, Hello under Macro Name.
3. Click the Create button to open the Visual Basic Editor with the outlines of a new macro already typed in.

VBA is a full-featured programming language with a correspondingly full-featured programming environment. This article examines only those tools that you use to get started with programming, and that excludes most of the tools in the Visual Basic Editor. With this caveat, close the Properties window on the left side of the Visual Basic Editor, and ignore the two dropdown lists that appear above the code.

Figure 2. Visual Basic Editor

The Visual Basic Editor contains the following code.

```vb
Sub Hello()
    ' Code goes here
End Sub
```

Sub stands for Subroutine, which you can define for now as "macro". Running the Hello macro runs any code that is between Sub Hello() and End Sub.
Now edit the macro so that it looks similar to the following code.

```vb
Sub Hello()
    MsgBox "Hello, world!"
End Sub
```

Go back to the Developer tab in Excel and click the Macros button again.

Select the Hello macro in the list that appears and then click Run to display a small message box that contains the text, "Hello, world!"

You just created and implemented custom VBA code in Excel. Click OK in the message box to close it and finish running the macro.

If the message box does not appear, check your macro security settings and restart Excel.

**Making Macros Accessible**

You can also get to the Macros dialog box from the View tab, but if you use a macro frequently, you might find it more convenient to access it with a keyboard shortcut or a Quick Access Toolbar button.

To create a button for the Hello macro on the Quick Access Toolbar, use the following procedure.

The following procedure describes how to make a button for a macro on the Quick Access Toolbar:

**To create a button for a macro on the Quick Access Toolbar**

1. Click the File tab.
2. Click Options to open the Excel Options dialog box, and then click Quick Access Toolbar.
3. In the list under Choose commands from:, choose Macros. Find the text that is similar to Book1!Hello in the list that appears and select that text.
4. Click the Add >> button to add the macro to the list on the right side, and then click the Modify... button to select a button image to associate with the macro.
5. Click OK. You should see your new button on the Quick Access Toolbar above the File tab.

Now you can quickly run your macro at any time without using the Developer tab—give it a try.

**A Real-World Example**

Suppose that you have a workbook that contains lists on a large number of worksheets and that you want to change the name of each worksheet to match the heading of the list on that worksheet. Not every worksheet has a list on it, but if it does, the heading is in cell B1, and if it does not, cell B1 is blank. The names of worksheets without lists should be left alone.

Ordinarily, this could be a complex task that involves looking at each worksheet to see if it has a list, copying the name if it does, clicking the worksheet tab, and then pasting in the new name. Instead of performing all of those steps manually, use Excel VBA to rename the sheets automatically.

**Learning about Objects**

To solve a VBA programming problem, you must first find out which objects the code will manipulate. To research that information, an essential tool is the Excel Object Model Reference, which is part of the Excel 2007 Developer Reference on the Microsoft Developer Network (MSDN).

These reference materials will be updated for Excel 2010 when it is publicly released, but the Excel 2007 Developer Reference is adequate for most Excel 2010 purposes.

**Figure 3. Excel Object Model Reference on MSDN**
The first step is to find out how to manipulate the particular objects that you need to work with to accomplish your task; for example, worksheets, worksheet names, cells, and cell contents. In Excel, there are at least two ways to approach the problem:

- Go directly to the Object Model Reference.
- Record some of the actions that you want to automate, see how the recorded code manipulates the objects, and then go to the Object Model Reference for more information.

Opinions vary on which approach is preferable, but for now, try using the Macro Recorder first.

**Using the Macro Recorder**

Sometimes a simple recorded macro is all you need; in these cases, you do not even have to look at the code. More often, recording alone is not enough. Instead, it is a starting point in the following process.

**To use the Macro Recorder as a starting point to your solution**

1. Record the actions that you want to code.
2. Review the code and find the lines that perform those actions.
3. Delete the rest of the code.
4. Modify the recorded code.
5. Add variables, control structures, and other code that the Macro Recorder cannot record.

Begin your investigation by recording a macro that renames a worksheet to **New Name**. You can then use the recorded macro to develop your own macro that renames multiple worksheets based on their contents.

**To record a macro that renames a worksheet**

1. Click **Record Macro** on the **Developer** tab.
2. Name the macro **RenameWorksheets**, rename Sheet1 to **New Name**, and click **Stop Recording**.
3. Go to the **Developer** or **View** tab, click the **Macros** button, and choose **Edit** to open the Visual Basic Editor.

The code in the Visual Basic Editor should look similar to the following.

```vb
Sub RenameWorksheets()

```

```
The first four lines after the `Sub` line are comments. Any line that begins with an apostrophe is a comment and has no effect on what the macro does. The main uses for comments are the following:

- To make the code easier to understand, not just for you, but for anyone else who might need to modify the code later.
- To temporarily disable a line of code (called commenting it out).

The four comments in this recorded macro serve neither purpose, so delete them.

The next line uses the `Select` method to select the Sheet1 member of the `Sheets` collection object. In VBA code, it is not generally necessary to select objects before manipulating them, even though that is what the Macro Recorder does. In other words, this line of code is redundant, so you can delete it as well.

The last line of the recorded macro modifies the Name Property of the `Sheet1` member of the `Sheets` collection. This is the line to keep.

After you make your changes, the recorded code should now look like the following.

```vb
Sub RenameWorksheets()
    Sheets("Sheet1").Name = "New Name"
End Sub
```

Manually change the sheet called New Name back to Sheet1, then run the macro. The name should change back to New Name.

### Modifying the Recorded Code

Now it is time to research the `Sheets` collection that the Macro Recorder used. The Sheets topic in the Object Model Reference includes the following text.

"The `Sheets` collection can contain `Chart` or `Worksheet` objects. If you need to work with sheets of only one type, see the object topic for that sheet type."

You are only working with `Worksheets`, so change the code to the following.

```vb
Sub RenameWorksheets()
    Worksheets("Sheet1").Name = "New Name"
End Sub
```

### Looping

One limitation of the code up to this point is that it only makes a change to one worksheet. You could add another line for each worksheet that you want to rename, but what if you do not know how many worksheets there are, or what their current names are? You need a way to apply some rule for each sheet in the workbook.

VBA has a construction called a For Each loop that is ideal. The For Each loop examines each item in a collection object such as `Worksheets` and can be used to take an action (like change a name) to some or all of those items.

For more information about the For Each loop, see the VBA Language Reference. Click "Visual Basic Conceptual Topics", then "Using For Each...Next Statements". Also, be aware that the VBA Language Reference, like the Object Model Reference, will amply repay the time that you spend browsing it, and is an excellent place to look for ideas if you get stuck working on code.

Using the third example in the "Using For Each...Next Statements" topic, edit the macro so that it looks similar to the following code.

```vb
Sub RenameWorksheets()
    For Each ws In Worksheets
        ws.Name = "New Name"
    Next
End Sub
```
myWorksheet is a variable; that is, what it represents varies. In this case, the myWorksheet variable successively represents each worksheet in the Worksheets collection. You do not have to use myWorksheet; you could use "x", "ws", "WorksheetToRenameAfterTheContentsOfCellB1", or (with a few restrictions) almost any name that you want. A good rule of thumb is to use variable names that are long enough to remind you of what the variable stands for, but not so long as to clutter the code.

If you run the macro in its current state, it produces an error because Excel requires each worksheet in a workbook to have a unique name, but the following line instructs Excel to give every worksheet the same name.

To correct the line so that you can verify that the For Each loop works, change the line to the following.

Instead of trying to give each worksheet the same name, this line changes the current name of each worksheet (myWorksheet.Name) to the current name with "-changed" appended to it.

**Useful Renaming**

The macro is getting close to something that might actually solve the problem at hand. What you need now is a way to take information from the worksheets themselves—specifically from cell B1 on each worksheet—and put that information into the names of the worksheets.

This time, instead of using the Macro Recorder to find out how to refer to a cell, take a guess and see if using the Cell object will work. It is a good guess, but if you open the Object Model Reference and search for the Cell object, you find that there is no Cell object! There is a CellFormat object though.

The CellFormat object topic includes the following code in the first code sample.

It turns out that you use Range to specify a range of cells or just one individual cell. Again, you do not need the .Select portion, but you do need to find out how to refer to the contents of the Range object, as opposed to the Range object itself. If you go to the Range object topic, you can read that Range has both Methods and Properties. The contents of a Range is a thing, not an action, so it would probably be a Property. If you scan down through the list, you can see the Value property. So, try the following.

You get an error if you run this on a workbook that contains worksheets where B1 is empty, because an empty Range has a Value of "" (an empty text string), which is not a legal worksheet name. It is about time to create some sample data anyway. Make the three sheets in the workbook look similar to the figure below, and then run the macro.
Figure 4. Sample data for the RenameWorksheets macro

<table>
<thead>
<tr>
<th>B1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>Occupation: Doctors</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

The worksheet names should change accordingly.

Checking for Empty Cells

As noted earlier, the macro fails if any of the B1 cells in the workbook are empty. Instead of manually checking every worksheet, you can code the macro to do it for you. Before the `myWorksheet.Name` line, add the following line of code.

```vb
If myWorksheet.Range("B1").Value <> "" Then
```

And after the `myWorksheet.Name` line add the following text.

```vb
End If
```

This is called an If...Then statement. The If...Then statement instructs Excel to do whatever is on the lines between the If line and the End If line, but only if the condition in the If line is met. In the example, the following line specifies the condition to meet.

```vb
myWorksheet.Range("B1").Value <> ""
```

The <> means "is not equal to", and the quotation marks with nothing between them represent an empty text string; that is, no text at all. Therefore, whatever lines of code come between the If and the End If will only be executed if the value in B1 is not equal to nothing; that is, if there is text in cell B1.

For more information about the If...Then statement, see the VBA Language Reference. (The full name is "If...Then...Else statement", where Else is an optional component.)

Variable Declarations

Another improvement that you should make to the macro is to put a declaration of the `myWorksheet` variable at the start of the macro.
Dim is short for "Dimension", and Worksheet is the type of this particular variable. This statement tells VBA what kind of entity myWorksheet represents. Note that after you type As, the Visual Basic Editor displays a popup that lists all the available variable types. That is an example of IntelliSense technology; that is, the Visual Basic Editor responds to what it determines you are trying to do and offers a list of appropriate options. You can choose an option from the list or just continue typing.

Although variable declarations are not required in VBA, using them is strongly recommended! Variable declarations make it much easier to keep track of your variables and to track down bugs in the code. Also, be aware that if you declare a variable with an object type (like Worksheet), IntelliSense displays an appropriate list of properties and methods associated with that object if you use the object variable later in the macro.

Comments

The macro is complex enough now to include some comments that remind you what the code is doing. The number of comments to use is partly a matter of personal style, but in general, too many comments are better than too few. Code usually needs to be modified and updated over time. Without comments, it can be hard to understand what is going on in the code, especially if the person who modifies the code is not the same person who wrote it in the first place. Adding comments for the If condition and for the line that renames the worksheets, results in the following code.

```
Sub RenameWorksheets()
    Dim myWorksheet As Worksheet
    For Each myWorksheet In Worksheets
        'make sure that cell B1 is not empty
        If myWorksheet.Range("B1").Value <> "" Then
            'rename the worksheet to the contents of cell B1
            myWorksheet.Name = myWorksheet.Range("B1").Value
        End If
    Next
End Sub
```

To test the macro, rename the worksheets back to Sheet1, Sheet2, and Sheet3 and delete the contents of cell B1 on one or more of the worksheets. Run the macro to verify that it renames the worksheets that have text in cell B1 and leaves the other worksheets alone. The macro works for any number of worksheets, with any combination of populated and empty B1 cells.

More Things that You Can Do with VBA

This section looks at a few more things that you can do with VBA in Excel 2010. The examples in this section are designed to give you a sense of the capabilities of VBA instead of focusing on specific real-world scenarios. You might find it helpful to review the information in the Object Model Reference about the objects in each step as you work through the examples.

Importance of Being Open

A good way to learn programming in general, and Excel VBA in particular, is a strategy where you try something, get it to work, and then ask yourself questions such as:

- What could I try next?
- What do I want to learn first, given how I want to use VBA?
- What might just be fun or interesting to know?
- What am I curious about?

The reader is strongly encouraged to investigate whatever avenues open up on the path to knowledge.

Charts

One common task in Excel is to create a chart based on a range of cells. Create a new macro called AssortedTasks and then type the following text in the Visual Basic Editor.
Add a line to create the chart object and assign the `myChart` variable to it.

```vbnet
Dim myChart As ChartObject
```

The numbers in the parentheses determine the position and size of the chart. The first two numbers are the coordinates of the top left corner, and the second two numbers are the width and height.

Create a new blank worksheet and run the macro. The chart it creates is useless because the chart has no data. Delete the chart that you just created, and add the following lines to the end of the macro.

```vbnet
With myChart
    .Chart.SetSourceData Source:=Selection
End With
```

This is a common pattern in VBA programming. First, you create an object and assign it to a variable, then you use the `With...End With` construction to do things with the object. The example code instructs the chart to use the current selection for its data. (`Selection` is a value for the `Source` parameter of the `SetSourceData` method, not a value of an object property, so VBA syntax requires that you use a colon and an equal sign (`:=`) instead of just an equal sign (`=`) to assign the value.)

Type some numbers in cells A1:A5, select the cells, and then run the macro. The chart shows up as the default type, a bar chart.

**Figure 5. Bar chart created by using VBA**

![Bar chart created by using VBA](http://msdn.microsoft.com/en-us/library/ee814737.aspx)

If you do not like the bar chart, you can change it to some other kind of chart by using code that is similar to the following.

```vbnet
...
xlPie is an example of a built-in constant, also known as an enumerated constant. There are many of these throughout Excel, and they are exhaustively documented. For more information about built-in constants, see the Enumerations section of the Object Model Reference. For example, the constants for chart types are listed under "XlChartType Enumeration".

You can modify the data. For example, try adding this line right after the variable declaration.

```
With myChart
   .Chart.SetSourceData Source:=Selection
   .Chart.ChartType = xlPie
End With
```

You can get input from the user, and use that input to modify the data.

```
myInput = InputBox("Please type a number:"
Application.ActiveSheet.Range("a5").Value = myInput
```

Finally, add the following lines to the end of the macro.

```
ActiveWorkbook.Save
ActiveWorkbook.Close
```

The complete macro should now look something like the following.

```
Sub AssortedTasks()
    Dim myChart As ChartObject
    Application.ActiveSheet.Range("a4").Value = 8
    myInput = InputBox("Please type a number:"
    Application.ActiveSheet.Range("a5").Value = myInput
    Set myChart = ActiveSheet.ChartObjects.Add(100, 50, 200, 200)
    With myChart
        .Chart.SetSourceData Source:=Selection
        .Chart.ChartType = xlPie
    End With
    ActiveWorkbook.Save
    ActiveWorkbook.Close
End Sub
```

Verify that cells A1:A5 are still selected, run the macro, type a number in the input box, and then click **OK**. The code saves and closes the workbook. Reopen the workbook and note the change to the pie chart.

**UserForms**

The previous section showed how to use a simple input box to get input from the user. In addition to the corresponding message box that displays information, VBA provides extensive capabilities that you can use to create custom dialog boxes, to code controls that are placed directly on worksheets, or to manipulate the dialog boxes that are built in to Excel. For more information about these capabilities, see Controls, Dialog Boxes, and Forms in the Excel 2007 Developer Reference.

This section completes this introductory look at Excel VBA by taking a quick look at UserForms.

On the **Developer** tab, click the **Visual Basic** button to open the Visual Basic Editor, and then go to the **Insert** menu and choose **UserForm** to open the UserForm Design View.

You will see two windows. One represents the UserForm that you are creating and the other, the **Toolbox**, shows the various controls that you can add to your UserForm; for example, command buttons, option buttons, check boxes, and
so on. You can move the mouse over a **Toolbox** control to see what type of control it creates.

Create a very simple UserForm with a single button that runs the Hello macro from earlier in this article. In the **Toolbox**, depress the the **CommandButton** control, and then drag it to the UserForm to create a command button. Right-click the command button and choose **View Code**.

The **Sub** that you see is a skeleton for an **event procedure**, which runs when a particular event occurs. In this case, as the name of the **Sub** indicates, the event that runs the code is a **Click** on **CommandButton1**. Add the following line to the event procedure.

```vb
Run("Hello")
```

The Visual Basic Editor should now look similar to the following.

**Figure 6. CommandButton1_Click event procedure**

Save the workbook, go to the **Window** menu, and choose **UserForm1 (UserForm)** to redisplay the UserForm. Click the green arrow on the toolbar to run the UserForm. When the dialog box appears, click the command button to run the Hello macro, which displays the "Hello, world!" message box. Close the message box to return to the running UserForm, then close the running UserForm to return to **Design View**.

**What's Next?**

You might find that the information in this article, when combined with some experimentation and quality time studying the Object Model Reference and the VBA Language Reference, gives you enough information to accomplish whatever task that motivated you to start learning VBA. If so, great! If not, a good next step is to broaden your scope to a more general understanding of VBA.

One way to learn more about VBA is to study working code. In addition to the examples in the Object Model Reference and the VBA Language Reference, there is a tremendous amount of Excel VBA code available from various online sources, including articles on MSDN, Web sites that are maintained by Microsoft Most Valuable Professionals (MVPs) who specialize in Excel, and others that you can find with a quick search of the Web.

The code in those resources can help you to solve your immediate coding problems and to give you ideas for projects that you might not have thought of yet.

If you prefer to pursue a more systematic study of VBA, there are several good books available on VBA, and several good reviews of each of those books on the Web that can help you to choose the best ones for your learning style.

**Additional Resources**

- Excel Developer Center
- Office Developer Center
- Excel MVPs at MSDN
- Excel MVPs at MVPs.org
- Getting Started with VBA in Office 2010
Community Content

Problem with FileSearch

Forgot to mention that my Excel is 2010 version and I understood that FileSearch is not supported any more.

1/4/2012
Sturenkatu

Problem with FileSearch

How can I replace FileSearch in the following piece of macro?

With Application.FileSearch
    .NewSearch
    .LookIn = sFolder
    .SearchSubFolders = False
    .Filename = ".xls"

    If .Execute() > 0 Then
        For i = 1 To .FoundFiles.Count
            sFile = .FoundFiles(i)

            ' Excel tiedoston nimessä pitää olla sana "lomake"
            If InStr(1, UCase(sFile), "LOMAKE", vbTextCompare) > 0 Then
                Set oWB = Workbooks.Open(sFile)
                ' Haetaan lomakkeelle syötetyt tiedot "autot" työkirjaan
                Row = 2
                While Not IsEmpty(oWB.Worksheets(1).Cells(Row, 2))
                    LastRow = LastRow + 1
                    Cells(LastRow, 1).Value = LastRow - 1
                    Cells(LastRow, 2).Value = oWB.Worksheets(1).Cells(Row, 2)
                    Cells(LastRow, 3).Value = oWB.Worksheets(1).Cells(Row, 3)
                    Cells(LastRow, 4).Value = oWB.Worksheets(1).Cells(Row, 4)
                    Row = Row + 1
                    Wend

                ' Suljetaan avattu lomake
                oWB.Close
                Set oWB = Nothing
            End If
        Next i
    End If
End With

1/4/2012
Sturenkatu

Problem with FileSearch

How can I replace FileSearch in the following piece of macro?
With Application.FileSearch
  .NewSearch
  .LookIn = sFolder
  .SearchSubFolders = False
  .Filename = ".xls"

  If .Execute() > 0 Then
    For i = 1 To .FoundFiles.Count
      sFile = .FoundFiles(i)

        ' Excel tiedoston nimessä pitää olla sana "lomake"
      If InStr(1, UCase(sFile), "LOMAKE", vbTextCompare) > 0 Then
        Set oWB = Workbooks.Open(sFile)

        ' Haetaan lomakkeelle syötetyt tiedot "autot" työkirjaan
        Row = 2
        While Not IsEmpty(oWB.Worksheets(1).Cells(Row, 2))
          LastRow = LastRow + 1
          Cells(LastRow, 1).Value = LastRow - 1
          Cells(LastRow, 2).Value = oWB.Worksheets(1).Cells(Row, 2)
          Cells(LastRow, 3).Value = oWB.Worksheets(1).Cells(Row, 3)
          Cells(LastRow, 4).Value = oWB.Worksheets(1).Cells(Row, 4)
          Row = Row + 1
        Wend

        ' Suljetaan avattu lomake
      oWB.Close
      Set oWB = Nothing
    End If
  Next i
End If
End With

HI

Hi, would you please help me... I am so confused....where can i ask my questions about VBA?

I have a question.... I want a VBA code for changing the color of my excel sheet cells

But i am using excel 2007

help help

Event - Prompt in Message Box

Hi is there somebody that can help me. I want to insert an event in an excel workbook to prompt the user for action in the form of a message box, but the event must run e.g. 10th day of the new month and if that day falls on a weekend the event must run the following wednesday of that month.

VBA copy sheet 2 to 1?

HI, Would require some help on displaying the second sheet data on the first sheet by clicking a button.
VBA Macro Problem

I have a problem with the following

a macro is called by the "Process Data" button,

The problem seems to be with the xlFilterCopy command

Range("Database").AdvancedFilter Action:=xlFilterCopy, _
   CriteriaRange:=Range("Criteria1"), _
   CopyToRange:=Range("Extract1"), _
   Unique:=False

NOTE: When the data range "Database" is manually copied into a new workbook, new sheet etc. Then the code will work, but when the whole worksheet "Data" is copied into a new workbook, the code hangs.

NOTE: When the range "Database" is changed so that it includes only the cells with data in them, the macro works.

Any ideas

How to call VBA Command Click in Excel Sheet

I have written a VBA code in Command Click, for eg HELlo, now how can i call this from Excel 2010, without come to VBA Editor?

Thank You

This was just soo helpful i want to thank you :D <3 xx

Excel Macros

I recorded a macro to add a footer. I also recorded a macro to remove the footer and it doesn't work. What is the secret? Thanks.

Examples don’t show good practices

Sub Hello()
   MsgBox("Hello, world!")
EndSub

Come on. Why are there parentheses following MsgBox?