Day 4

1pm-2pm

PROCEDURES

Introduction to Procedures

Procedures

Introduction to Sub-Procedures

Calling a Sub Procedure

Procedures and Access Levels

Declaring a Sub procedure

A procedure declared with the Sub keyword must adhere to the following syntax:

[Private | Public][Static] Sub name ([arglist])

[instructions]

[Exit Sub]

[instructions]

End Sub

Here’s a description of the elements that make up a Sub procedure:

* Private: Optional. Indicates that the procedure is accessible only to other procedures in the same module.
* Public: Optional. Indicates that the procedure is accessible to all other procedures in all other modules in the workbook. If used in a module that contains an Option
* Private Module statement, the procedure is not available outside the project (other workbooks or Microsoft Office applications that may attempt to call the procedures in the module).
* Static: Optional. Indicates that the procedure’s variables are preserved when the procedure ends.
* Sub: Required. The keyword that indicates the beginning of a procedure.
* name: Required. Any valid procedure name.
* arglist: Optional. Represents a list of variables, enclosed in parentheses, that receive arguments passed to the procedure. Use a comma to separate arguments. If the procedure uses no arguments, a set of empty parentheses is required.
* instructions: Optional. Represents valid VBA instructions.
* Exit Sub: Optional. Forces an immediate exit from the procedure prior to its formal completion.
* End Sub: Required. Indicates the end of the procedure.

Executing Sub Procedures

* You can call Sub procedure with the Run ➪ Run Sub/UserForm command (in the VBE menu). You can also call a Sub procedure by pressing the F5 shortcut key, or you can click the Run Sub/UserForm button on the Standard toolbar. These methods all assume that the cursor is within a procedure.
* You can call a Sub procedure from Excel’s Macro dialog box.
* You can call a Sub procedure by using the Ctrl key shortcut assigned to the procedure (assuming that you assigned one).
* You can call a Sub procedure by clicking a button or shape assigned to the procedure.
* You can specify that a Sub procedure be run when an event occurs, such as opening the workbook, saving the workbook, closing the workbook, changing a cell’s value, or activating a sheet.

Private Sub Worksheet\_Activate()

Call greetme

End Sub

* You can call a Sub procedure from another procedure that you write. Sub and Function procedures can execute other procedures.

Sub greetme()

MsgBox “hello”

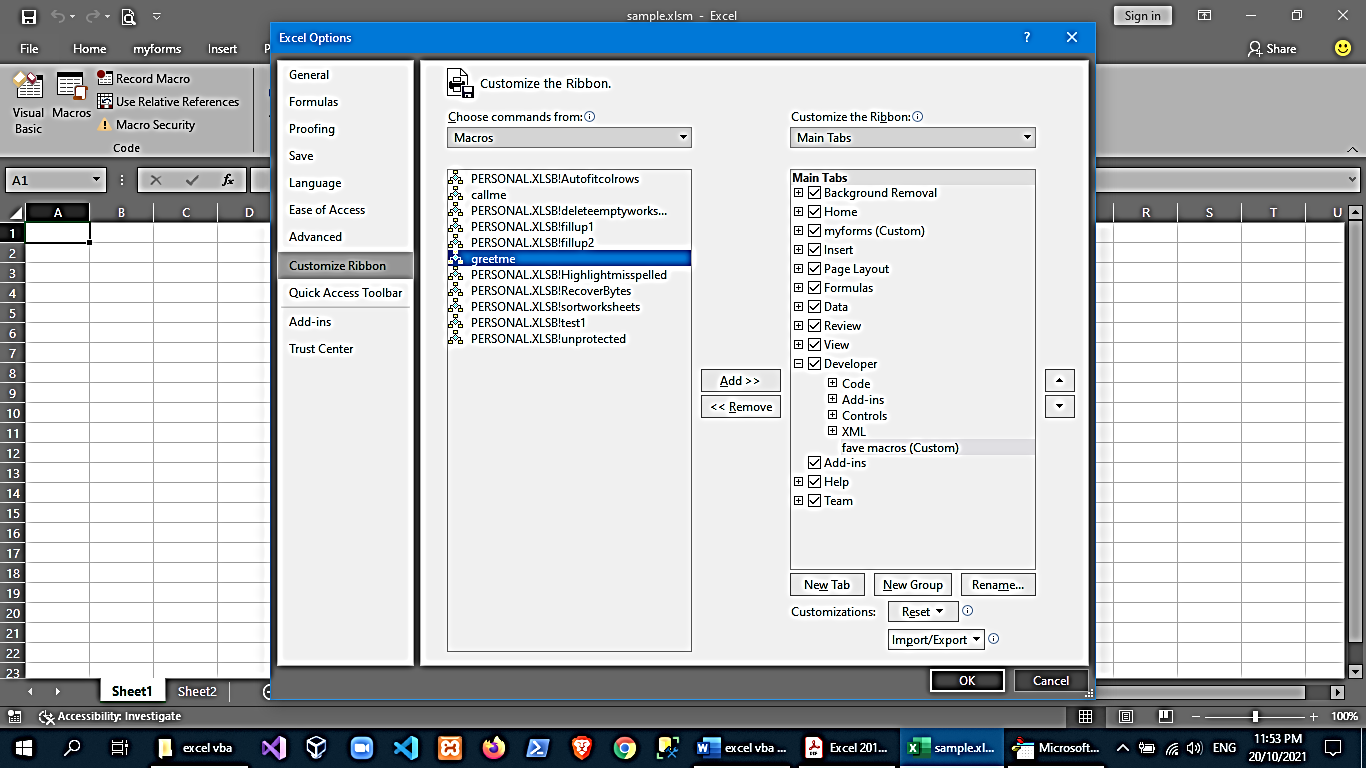
End Sub

Sub callme()

Call greetme

End Sub

* You can call a Sub procedure from an icon added to the Quick Access toolbar.



* You can call a Sub procedure from a customized shortcut menu.

1. Put the following code in the ThisWorkbook code window

Private Sub Workbook\_Open()

Dim MyMenu As Object

Set MyMenu = Application.ShortcutMenus(xlWorksheetCell).MenuItems.AddMenu(“This is my Custom Menu”, 1)

With MyMenu.MenuItems

.Add “MyMacro1”, “MyMacro1”, , 1, , ““

.Add “MyMacro2”, “MyMacro2”, , 2, , ““

End With

Set MyMenu = Nothing

End Sub

1. This is the sample macro code in a module

Public Sub mymacro1()

MsgBox “Macro1 from a right click menu”

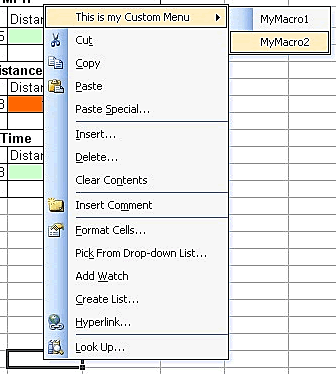
End Sub

Public Sub mymacro2()

MsgBox “Macro2 from a right click menu”

End Sub

1. Close your workbook and re-open!



example 1:

‘macro to delete an entire row (added to shortcut menu)

Sub Test()

Dim c As Range, myRange As Range

On Error Resume Next

Set myRange = Sheets(2).Range(“A1:A100”)

For Each c In myRange

If c.Value = ActiveCell.Value Then

c.EntireRow.Delete

End If

Next c

ActiveCell.EntireRow.Delete

End Sub

Private Sub Workbook\_Open()

Dim MyMenu As Object

Set MyMenu = Application.ShortcutMenus(xlWorksheetCell).MenuItems.AddMenu(“A minha Macro”, 1)

With MyMenu.MenuItems

.Add “Test”, “Test”, , 1, , ““

End With

Set MyMenu = Nothing

End Sub

example 2:

‘another way of adding an item to shortcut menu

‘select column A of all worksheets

Sub SelectA1()

Dim sht As Worksheet, csheet As Worksheet

Application.ScreenUpdating = False

Set csheet = ActiveSheet

For Each sht In ActiveWorkbook.Worksheets

If sht.Visible Then

sht.Activate

Range(“A:A”).Select

ActiveWindow.ScrollRow = 1

ActiveWindow.ScrollColumn = 1

End If

Next sht

csheet.Activate

Application.ScreenUpdating = True

End Sub

Sub AddToShortCut()

Dim Bar As CommandBar

Dim NewControl As CommandBarButton

DeleteFromShortcut

Set Bar = Application.CommandBars(“Cell”)

Set NewControl = Bar.Controls.Add \_

(Type:=msoControlButton, ID:=1, \_

temporary:=True)

With NewControl

.Caption = “&Select all A1”

.OnAction = “SelectA1”

.Style = msoButtonIconAndCaption

End With

End Sub

Sub DeleteFromShortcut()

On Error Resume Next

Application.CommandBars(“Cell”).Controls \_

(“&Change Case”).Delete

End Sub

‘add to *thisworkbook* code

Private Sub Workbook\_Open()

Call AddToShortCut

End Sub

Private Sub Workbook\_BeforeClose(Cancel As Boolean)

Call DeleteFromShortcut

End Sub

* Finally, you can run a Sub procedure from the Immediate window in VBE. Just type the name of the procedure, including any arguments that may apply, and press Enter. To display Immediate Window, in VBE🡪view🡪immediate window

2pm-3pm

DATES AND TIMES IN VBA EXCEL

Fundamentals of Dates

The Components of a Date

Formatting a Date Value

Built-In Time Functions

The Components of a Time Value

Operations on Date and Time Values

‘date function

Sub UsingTheDateFunction()

Dim theDate As Date

theDate = Date()

Debug.Print theDate

End Sub

‘now function

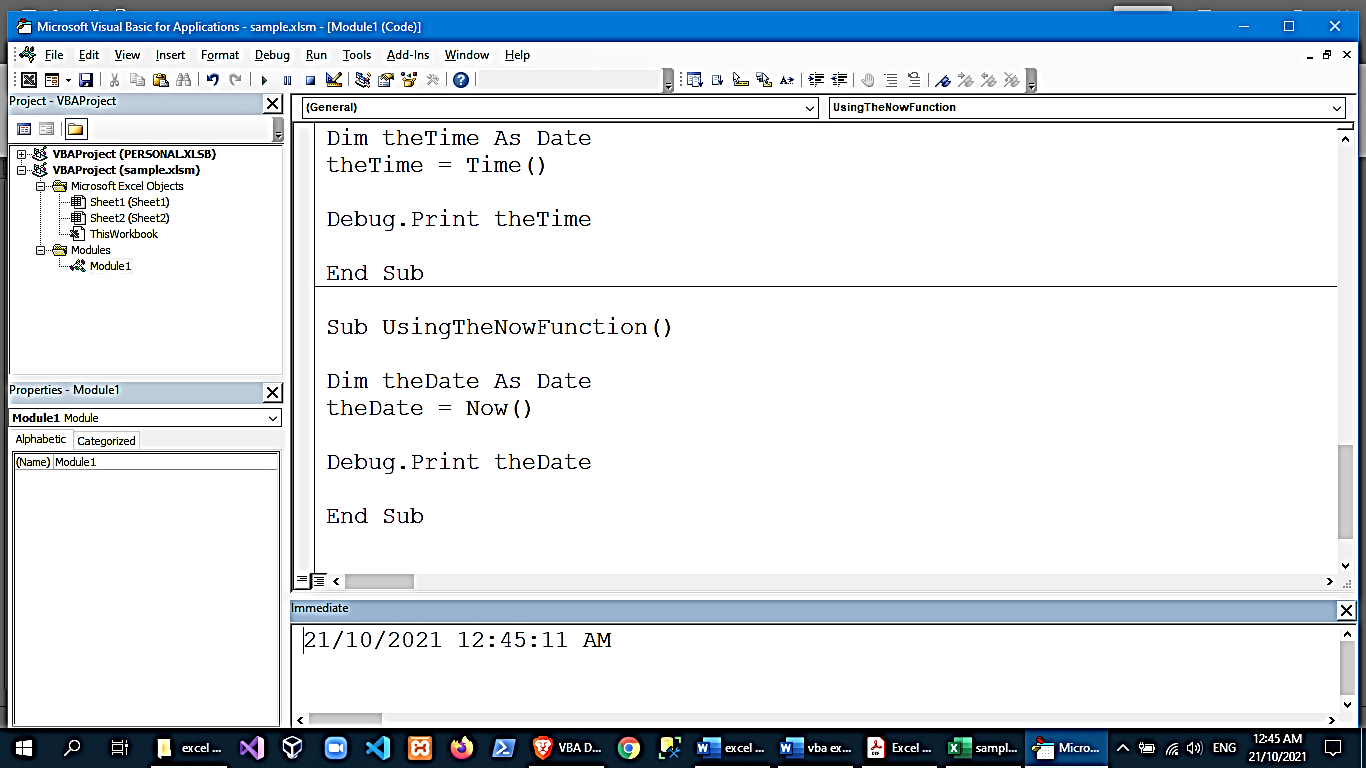
Sub UsingTheNowFunction()

Dim theDate As Date

theDate = Now()

Debug.Print theDate

End Sub



‘time function

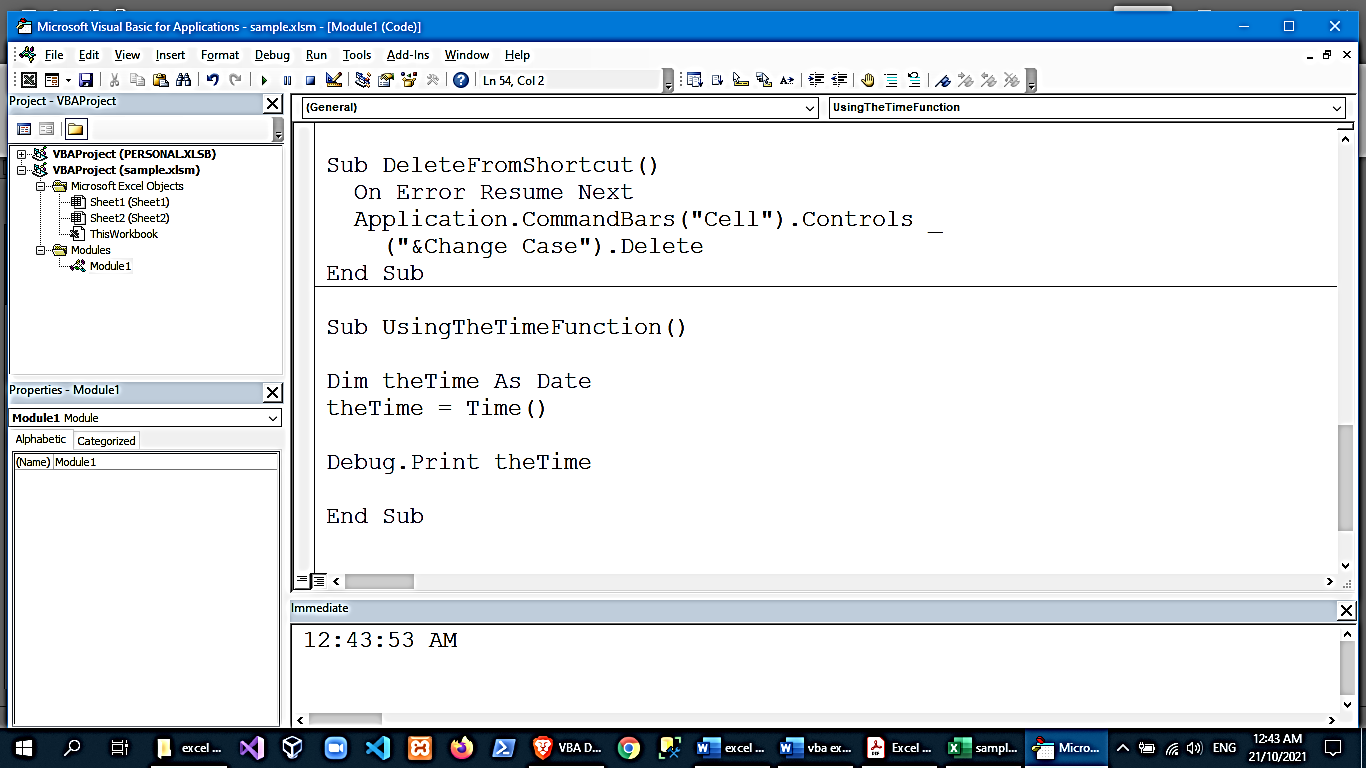
Sub UsingTheTimeFunction()

Dim theTime As Date

theTime = Time()

Debug.Print theTime

End Sub



**VBA DateAdd Function**

You can use the DateAdd Function to add a date/time interval to a date or time, and the function will return the resulting date/time.

The syntax of the DateAdd Function is:

DateAdd(Interval, Number, Date) where:

Interval – A string that specifies the type of interval to use. The interval can be one of the following values:

“d” – day

“ww” – week

“w” – weekday

“m” – month

“q” – quarter

“yyyy” – year

“y” – day of the year

“h” – hour

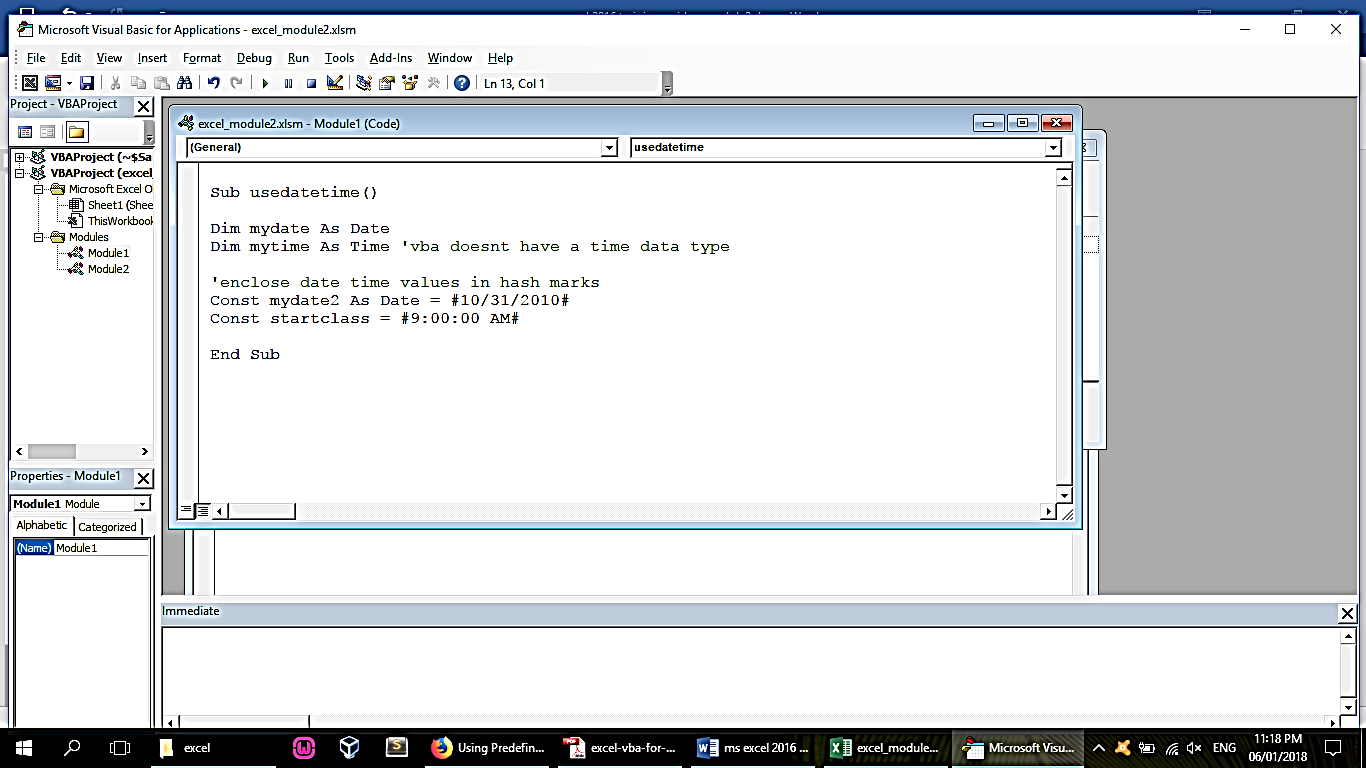
“n” – minute

“s” – second

Number – The number of intervals that you want to add to the original date/time.

Date – The original date/time.

Note: When using dates in your code you have to surround them with # or quotation marks.



The following code shows how to use the DateAdd Function:

Sub UsingTheDateAddFunction()

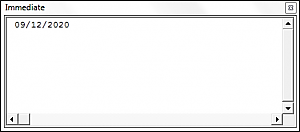
Dim laterDate As Date

laterDate = DateAdd(“m”, 10, “11/12/2019”)

Debug.Print laterDate

End Sub

The result is:



**VBA DateDiff Function**

You can use the DateDiff Function in order to get the difference between two dates, based on a specified time interval.

The syntax of the DateDiff Function is:

DateDiff(Interval, Date1, Date2, [Firstdayofweek], [Firstweekofyear]) where:

Interval – A string that specifies the type of interval to use. The interval can be one of the following values:

“d” – day

“ww” – week

“w” – weekday

“m” – month

“q” – quarter

“yyyy” – year

“y” – day of the year

“h” – hour

“n” – minute

“s” – second

Date1 – A date value representing the earlier date.

Date2 – A date value representing the later date.

Firstdayofweek (Optional) – A constant that specifies the weekday that the function should use as the first day of the week. If blank Sunday is used as the first day of the week. Firstdayofweek can be one of the following values:

-vbSunday – uses Sunday as the first day of the week.

-vbMonday – uses Monday as the first day of the week.

-vbTuesday – uses Tuesday as the first day of the week.

-vbWednesday – uses Wednesday as the first day of the week.

-vbThursday – uses Thursday as the first day of the week.

-vbFriday – uses Friday as the first day of the week.

-vbSaturday – uses Saturday as the first day of the week.

-vbUseSystemDayOfTheWeek – uses the first day of the week that is specified by your system’s settings.

Firstweekofyear (Optional) – A constant that specifies the first week of the year. If blank then the Jan 1st week is used as the first week of the year. Firstweekofyear can be one of the following values:

-vbFirstJan1 – uses the week containing Jan 1st.

-vbFirstFourDays – uses the first week that contains at least four days in the new year.

-vbFirstFullWeek – uses the first full week of the year.

-vbSystem – uses the first week of the year as specified by your system settings.

The following code shows you how to use the DateDiff Function:

Sub UsingTheDateDiffFunction()

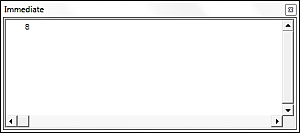
Dim theDifferenceBetweenTwoDates As Long

theDifferenceBetweenTwoDates = DateDiff("q", "11/11/2010", "10/12/2012")

Debug.Print theDifferenceBetweenTwoDates

End Sub

The result is:



**VBA DatePart Function**

You can use the DatePart Function in order to return a part (day, week, quarter, month etc.) of a given date.

The syntax of the DatePart Function is:

DatePart(Interval, Date,[Firstdayofweek], [Firstweekofyear]) where:

Interval – A string that specifies the part of the date to return. The interval can be one of the following values:

“d” – day

“ww” – week

“w” – weekday

“m” – month

“q” – quarter

“yyyy” – year

“y” – day of the year

“h” – hour

“n” – minute

“s” – second

Date – The date that you want the function to return a part of.

Firstdayofweek (Optional) – A constant that specifies the weekday that the function should use as the first day of the week. If blank Sunday is used as the first day of the week. Firstdayofweek can be one of the following values:

-vbSunday – uses Sunday as the first day of the week.

-vbMonday – uses Monday as the first day of the week.

-vbTuesday – uses Tuesday as the first day of the week.

-vbWednesday – uses Wednesday as the first day of the week.

-vbThursday – uses Thursday as the first day of the week.

-vbFriday – uses Friday as the first day of the week.

-vbSaturday – uses Saturday as the first day of the week.

-vbUseSystemDayOfTheWeek – uses the first day of the week that is specified by your system’s settings.

Firstweekofyear (Optional) – A constant that specifies the first week of the year. If blank then the Jan 1st week is used as the first week of the year. Firstweekofyear can be one of the following values:

-vbFirstJan1 – uses the week containing Jan 1st.

-vbFirstFourDays – uses the first week that contains at least four days in the new year.

-vbFirstFullWeek – uses the first full week of the year.

-vbSystem – uses the first week of the year as specified by your system settings.

The following code shows you how to use the DatePart Function:

Sub UsingTheDatePartFunction()

Dim thePartOfTheDate As Integer

thePartOfTheDate = DatePart("yyyy", "12/12/2009")

Debug.Print thePartOfTheDate

End Sub



**VBA DateSerial Function**

The VBA DateSerial Function takes an input year, month and day and returns a date.

The syntax of the DateSerial Function is:

DateSerial(Year, Month, Day) where:

Year – An integer value between 100 and 9999 that represents the year.

Month – An integer value that represents the month.

Day – An integer value that represents the day.

The following code shows you how to use the DateSerial Function:

Sub UsingTheDateSerialFunction()

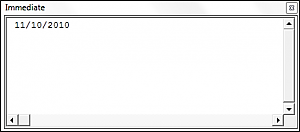
Dim theDate As Date

theDate = DateSerial(2010, 11, 10)

Debug.Print theDate

End Sub

The result is:



**VBA DateValue Function**

The DateValue Function returns a Date when given a string representation of a date.

The syntax of the DateValue Function is:

DateValue(Date) where:

Date – A String representing the date.

The following code shows you how to use the DateValue Function:

Sub UsingTheDateValueFunction()

Dim theDate As Date

theDate = DateValue("October, 29, 2010")

Debug.Print theDate

End Sub

The result is:



‘other date and time functions

theDay = Day("10/12/2010")

theHour = Hour("2:14:17 AM")

theMinuteValue = Minute("2:14:17 AM")

theSecondValue = Second("2:14:17 AM")

theMonth = Month("11/18/2010")

theMonthName = MonthName(12, True) ‘*true* sets the month name to the abbreviated version

theTime = TimeSerial(1, 10, 15) ‘generates time format based on 3 inputs

theTime = TimeValue("22:10:17") ‘gets the time value from a string

theWeekDay = Weekday("11/20/2019")

theWeekdayName = WeekdayName(4)

theYear = Year("11/12/2010")

**Comparing Dates in VBA**

You can compare dates using the >, <, and = operators in VBA. The following code shows you how to compare two dates in VBA.

Sub ComparingDates()

Dim dateOne As Date

Dim dateTwo As Date

dateOne = "10/10/2010"

dateTwo = "11/11/2010"

If dateOne > dateTwo Then

Debug.Print "dateOne is the later date"

ElseIf dateOne = dateTwo Then

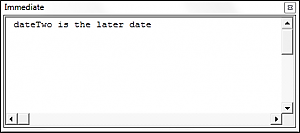
Debug.Print "The two dates are equal"

Else

Debug.Print "dateTwo is the later date"

End If

End Sub



3pm-4pm

ERROR HANDLING

Handling Errors

In Case of Error

Types of Error

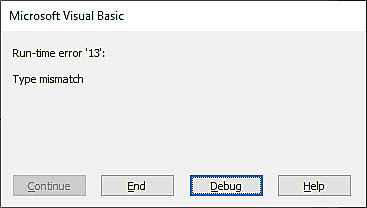
The Err Object

**On Error GoTo 0**

On Error GoTo 0 is VBA’s default setting. You can restore this default setting by adding the following line of code:

On Error GoTo 0

When an error occurs with On Error GoTo 0, VBA will stop executing code and display its standard error message box.



Often you will add an On Error GoTo 0 after adding On Error Resume Next error handling (next section):

Sub ErrorGoTo0()

On Error Resume Next

ActiveSheet.Shapes("Start\_Button").Delete

On Error GoTo 0

'Run More Code

End Sub

**On Error Resume Next**

On Error Resume Next tells VBA to skip any lines of code containing errors and proceed to the next line.

On Error Resume Next

Note: On Error Resume Next does not fix an error, or otherwise resolve it. It simply tells VBA to proceed as if the line of code containing the error did not exist. Improper use of On Error Resume Next can result in unintended consequences.

A great time to use On Error Resume Next is when working with objects that may or may not exist. For example, you want to write some code that will delete a shape, but if you run the code when the shape is already deleted, VBA will throw an error. Instead you can use On Error Resume Next to tell VBA to delete the shape if it exists.

…

On Error Resume Next

ActiveSheet.Shapes("Start\_Button").Delete

On Error GoTo 0

…

Notice we added On Error GoTo 0 after the line of code containing the potential error. This resets the error handling.

**Err.Number, Err.Clear, and Catching Errors**

Instead of simply skipping over a line containing an error, we can catch the error by using On Error Resume Next and Err.Number.

Err.Number returns an error number corresponding with the type of error detected. If there is no error, Err.Number = 0.

For example, this procedure will return “11” because the error that occurs is Run-time error ’11’.

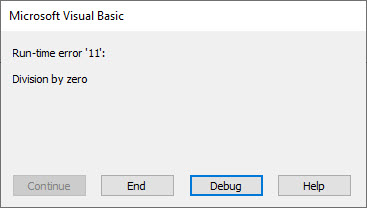
Sub ErrorNumber\_ex()

On Error Resume Next

ActiveCell.Value = 2 / 0

MsgBox Err.Number

End Sub



**Error Handling with Err.Number**

The true power of Err.Number lies in the ability to detect if an error occurred (Err.Number <> 0). In the example below, we’ve created a function that will test if a sheet exists by using Err.Number.

Sub TestWS()

MsgBox DoesWSExist("test")

End Sub

Function DoesWSExist(wsName As String) As Boolean

Dim ws As Worksheet

On Error Resume Next

Set ws = Sheets(wsName)

'If Error WS Does not exist

If Err.Number <> 0 Then

DoesWSExist = False

Else

DoesWSExist = True

End If

On Error GoTo -1

End Function

Note: We’ve added a On Error GoTo -1 to the end which resets Err.Number to 0.

With On Error Resume Next and Err.Number, you can replicate the “Try” & “Catch” functionality of other programming languages.

**On Error GoTo Line**

On Error GoTo Line tells VBA to “go to” a labeled line of code when an error is encountered. You declare the Go To statement like this (where errHandler is the line label to go to):

On Error GoTo errHandler

and create a line label like this:

errHandler:

Note: This is the same label that you’d use with a regular VBA GoTo Statement.

**On Error Exit Sub**

You can use On Error GoTo Line to exit a sub when an error occurs.

You can do this by placing the error handler line label at the end of your procedure:

Sub ErrGoToEnd()

On Error GoTo endProc

'Some Code

endProc:

End Sub

or by using the Exit Sub command:

Sub ErrGoToEnd()

On Error GoTo endProc

'Some Code

GoTo skipExit

endProc:

Exit Sub

skipExit:

'Some More Code

End Sub

**Err.Clear, On Error GoTo -1, and Resetting Err.Number**

After an error is handled, you should generally clear the error to prevent future issues with error handling.

After an error occurs, both Err.Clear and On Error GoTo -1 can be used to reset Err.Number to 0. But there is one very important difference: Err.Clear does not reset the actual error itself, it only resets the Err.Number.

What does that mean? Using Err.Clear, you will not be able to change the error handling setting. To see the difference, test out this code and replace On Error GoTo -1 with Err.Clear:

Sub ErrExamples()

On Error GoTo errHandler:

'"Application-defined" error

Error (13)

Exit Sub

errHandler:

' Clear Error

On Error GoTo -1

On Error GoTo errHandler2:

'"Type mismatch" error

Error (1034)

Exit Sub

errHandler2:

Debug.Print Err.Description

End Sub

Typically, I recommend always using On Error GoTo -1, unless you have a good reason to use Err.Clear instead.

**VBA On Error MsgBox**

You might also want to display a Message Box on error. This example will display different message boxes depending on where the error occurs:

Sub ErrorMessageEx()

Dim errMsg As String

On Error GoTo errHandler

'Stage 1

errMsg = "An error occurred during the Copy & Paste stage."

'Err.Raise (11)

'Stage 2

errMsg = "An error occurred during the Data Validation stage."

'Err.Raise (11)

'Stage 3

errMsg = "An error occurred during the P&L-Building and Copy-Over stage."

Err.Raise (11)

'Stage 4

errMsg = "An error occurred while attempting to log the Import on the Setup Page"

'Err.Raise (11)

GoTo endProc

errHandler:

MsgBox errMsg

endProc:

End Sub

Here you would replace Err.Raise(11) with your actual code.

**VBA IsError**

Another way to handle errors is to test for them with the VBA IsError Function. The IsError function tests an expression for errors, returning TRUE or FALSE if an error occurs.

Sub IsErrorEx()

MsgBox IsError(Range("a7").Value)

End Sub

**If Error Excel Function**

You can also handle errors in VBA with the Excel IfError Function. The IfError Function must be accessed by using the WorksheetFunction Class:

Sub IfErrorEx()

Dim n As Long

n = WorksheetFunction.IfError(Range("a10").Value, 0)

MsgBox n

End Sub

This will output the value of Range A10, if the value is an error, it will output 0 instead.

4pm-5pm

More Macro examples

**Insert Multiple Columns**

Sub InsertMultipleColumns()

Dim i As Integer

Dim j As Integer

ActiveCell.EntireColumn.Select

On Error GoTo Last

i = InputBox("Enter number of columns to insert", "Insert Columns")

For j = 1 To i

Selection.Insert Shift:=xlToRight, CopyOrigin:=xlFormatFromRightorAbove

Next j

Last: Exit Sub

End Sub

**Insert Multiple Rows**

Sub InsertMultipleRows()

Dim i As Integer

Dim j As Integer

ActiveCell.EntireRow.Select

On Error GoTo Last

i = InputBox("Enter number of columns to insert", "Insert Columns")

For j = 1 To i

Selection.Insert Shift:=xlToDown, CopyOrigin:=xlFormatFromRightorAbove

Next j

Last: Exit Sub

End Sub

**Remove Text Wrap**

Sub RemoveTextWrap()

Range("A1").WrapText = False

End Sub

**Unmerge Cells**

Sub UnmergeCells()

Selection.UnMerge

End Sub

**Highlight Duplicates from Selection**

Sub HighlightDuplicateValues()

Dim myRange As Range

Dim myCell As Range

Set myRange = Selection

For Each myCell In myRange

If WorksheetFunction.CountIf(myRange, myCell.Value) > 1 Then

myCell.Interior.ColorIndex = 36

End If

Next myCell

End Sub

**Highlight Top 10 Values**

Sub TopTen()

Selection.FormatConditions.AddTop10

Selection.FormatConditions(Selection.FormatConditions.Count).S

tFirstPriority

With Selection.FormatConditions(1)

.TopBottom = xlTop10Top

.Rank = 10

.Percent = False

End With

With Selection.FormatConditions(1).Font

.Color = -16752384

.TintAndShade = 0

End With

With Selection.FormatConditions(1).Interior

.PatternColorIndex = xlAutomatic

.Color = 13561798

.TintAndShade = 0

End With

Selection.FormatConditions(1).StopIfTrue = False

End Sub

**Highlight Greater than Values**

Sub HighlightGreaterThanValues()

Dim i As Integer

i = InputBox("Enter Greater Than Value", "Enter Value")

Selection.FormatConditions.Delete

Selection.FormatConditions.Add Type:=xlCellValue, \_

Operator:=xlGreater, Formula1:=i

Selection.FormatConditions(Selection.FormatConditions.Count).S

tFirstPriority

With Selection.FormatConditions(1)

.Font.Color = RGB(0, 0, 0)

.Interior.Color = RGB(31, 218, 154)

End With

End Sub

**Highlight Lower Than Values**

Sub HighlightLowerThanValues()

Dim i As Integer

i = InputBox("Enter Lower Than Value", "Enter Value")

Selection.FormatConditions.Delete

Selection.FormatConditions.Add \_

Type:=xlCellValue, \_

Operator:=xlLower, \_

Formula1:=i

Selection.FormatConditions(Selection.FormatConditions.Count).S

tFirstPriority

With Selection.FormatConditions(1)

.Font.Color = RGB(0, 0, 0)

.Interior.Color = RGB(217, 83, 79)

End With

End Sub

**Highlight Negative Numbers**

Sub highlightNegativeNumbers()

Dim Rng As Range

For Each Rng In Selection

If WorksheetFunction.IsNumber(Rng) Then

If Rng.Value < 0 Then

Rng.Font.Color= -16776961

End If

End If

Next

End Sub

**Highlight Cells with Comments**

Sub highlightCommentCells()

Selection.SpecialCells(xlCellTypeComments).Select

Selection.Style= "Note"

End Sub

**Highlight all the Blank Cells Invisible Space**

Sub blankWithSpace()

Dim rng As Range

For Each rng In ActiveSheet.UsedRange

If rng.Value = " " Then

rng.Style = "Note"

End If

Next rng

End Sub

Sometimes there are some cells which are blank but they have a single space and due to this, it’s really hard to identify them. This code will check all the cell in the worksheet and highlight all the cells which have a single space.

**Protect all Worksheets Instantly**

Sub ProtectAllWorskeets()

Dim ws As Worksheet

Dim ps As String

ps = InputBox("Enter a Password.", vbOKCancel)

For Each ws In ActiveWorkbook.Worksheets

ws.Protect Password:=ps

Next ws

End Sub

**Un-Protect Worksheet**

Sub UnprotectWS()

ActiveSheet.Unprotect "mypassword"

End Sub

**Protect all the Cells With Formulas**

Sub lockCellsWithFormulas()

With ActiveSheet

.Unprotect

.Cells.Locked = False

.Cells.SpecialCells(xlCellTypeFormulas).Locked = True

.Protect AllowDeletingRows:=True

End With

End Sub

**Save Each Worksheet as a Single PDF**

Sub SaveWorkshetAsPDF()

Dimws As Worksheet

For Each ws In Worksheets

ws.ExportAsFixedFormat \_

xlTypePDF, \_

"ENTER-FOLDER-NAME-HERE" & \_

ws.Name & ".pdf"

Next ws

End Sub

This code will simply save all the worksheets in a separate PDF file. You just need to change the folder name from the code.

**Create a Backup of a Current Workbook**

Sub FileBackUp()

ThisWorkbook.SaveCopyAs Filename:=ThisWorkbook.Path & \_

"" & Format(Date, "mm-dd-yy") & " " & \_

ThisWorkbook.name

End Sub

This is one of the most useful macros which can help you to save a backup file of your current workbook. It will save a backup file in the same directory where your current file is saved and it will also add the current date with the name of the file.

**Convert Range into an Image**

Sub PasteAsPicture()

Application.CutCopyMode = False

Selection.Copy

ActiveSheet.Pictures.Paste.Select

End Sub

**Use Text to Speech**

Sub Speak()

Selection.Speak

End Sub

Just select a range and run this code. Excel will speak all the text what you have in that range, cell by cell.

**Activate Data Entry Form**

Sub DataForm()

ActiveSheet.ShowDataForm

End Sub

There is a default data entry form which you can use for data entry.

**Convert all Formulas into Values**

Sub convertToValues()

Dim MyRange As Range

Dim MyCell As Range

Select Case \_

MsgBox("You Can't Undo This Action. " & "Save Workbook First?", vbYesNoCancel, "Alert")

Case Is = vbYes

ThisWorkbook.Save

Case Is = vbCancel

Exit Sub

End Select

Set MyRange = Selection

For Each MyCell In MyRange

If MyCell.HasFormula Then

MyCell.Formula = MyCell.Value

End If

Next MyCell

End Sub

Simply convert formulas into values. When you run this macro it will quickly change the formulas into absolute values.

**Remove Spaces from Selected Cells**

Sub RemoveSpaces()

Dim myRange As Range

Dim myCell As Range

Select Case MsgBox("You Can't Undo This Action. " & "Save Workbook First?", vbYesNoCancel, "Alert")

Case Is = vbYesThisWorkbook.Save

Case Is = vbCancel

Exit Sub

End Select

Set myRange = Selection

For Each myCell In myRange

If Not IsEmpty(myCell) Then

myCell = Trim(myCell)

End If

Next myCell

End Sub

One of the most useful macros from this list. It will check your selection and then remove all the extra spaces from that.

**Convert to Upper Case**

Sub convertUpperCase()

Dim Rng As Range

For Each Rng In Selection

If Application.WorksheetFunction.IsText(Rng) Then

Rng.Value = UCase(Rng)

End If

Next

End Sub

**Convert to Lower Case**

Sub convertLowerCase()

Dim Rng As Range

For Each Rng In Selection

If Application.WorksheetFunction.IsText(Rng) Then

Rng.Value= LCase(Rng)

End If

Next

End Sub

**Convert to Proper Case**

Sub convertProperCase()

Dim Rng As Range

For Each Rng In Selection

If WorksheetFunction.IsText(Rng) Then

Rng.Value = WorksheetFunction.Proper(Rng.Value)

End If

Next

End Sub

**Convert to Sentence Case**

Sub convertTextCase()

Dim Rng As Range

For Each Rng In Selection

If WorksheetFunction.IsText(Rng) Then

Rng.Value = UCase(Left(Rng, 1)) & LCase(Right(Rng, Len(Rng) - 1))

End If

Next Rng

End Sub