The European Journal of Education and Applied Psychology 2025, No 4
ISSN 2310-5704

PREMIER Publishing
ppublishing.org

DOI:10.29013/EJEAP-25-4-8-15

TEACHING METHODOLOGY FOR MICROSOFT EXCEL WORKBOOK CREATION USING THE EXCEL OBJECT MODEL (VBA)

Bashirova Goncha Imanverdi giziv [1]

[1] Azerbaijan State Pedagogical University

Cite: *Bashirova G.I. (2025). Teaching Methodology for Microsoft Excel Workbook Creation Using the Excel Object Model (VBA). European Journal of Education and Applied Psychology 2025, No 4.* https://doi.org/10.29013/EJEAP-25-4-8-15

Abstract

This article examines the object-oriented components of the Microsoft Excel Application Software System and provides foundational knowledge for establishing a programming interface between Visual Basic for Applications (VBA) and Excel. It is noteworthy that the Microsoft Excel Object Model (MEOM) displays certain differences from standard Object-Oriented Programming (OOP) paradigms. Programmers must consider these distinctions when developing application systems in the Excel and VBA environment. This study demonstrates how OOP mechanisms are implemented within MEOM and emphasizes its unique features. An Excel application file, referred to as a workbook, can be extensively manipulated through MEOM using VBA. The article introduces the primary components of MEOM that facilitate these operations and illustrates their application in practical procedures and functions.

Keywords: *MEOM, Delimiter, Editable, Converter, Notify, Origin, Corrupt Load, AddToMru.*

1. Introduction

Microsoft Excel is widely recognized as one of the most essential tools in modern business, academia, and data analysis. Beyond its familiar interface for spreadsheet calculations, Excel hosts a robust programming environment through Visual Basic for Applications (VBA), enabling users to automate tasks, develop custom functions, and build integrated application systems (Evensen, H.T., 2014). Central to this capability is the Microsoft Excel Object Model (MEOM)—a structured, hierarchical framework that exposes Excel's elements – such as workbooks, worksheets, ranges, charts, and even application-level settings – as programmable objects.

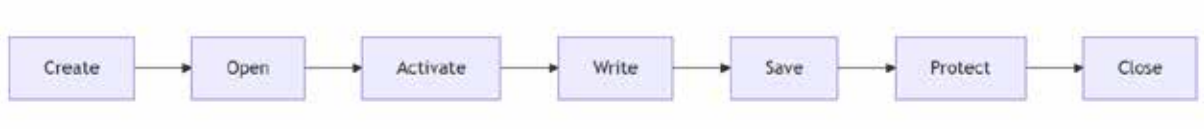
Unlike traditional object-oriented programming (OOP) environments, MEOM follows a slightly adapted model tailored to the spreadsheet paradigm, where objects often represent both data containers and interface elements. Understanding MEOM is fundamental for anyone seeking to move beyond manual Excel use into the realm of automated, scalable, and repeatable data processing. Through MEOM, developers can programmatically control every aspect of Excel – from creating and formatting workbooks to implementing complex data operations, applying security measures, and facilitating interaction between multiple files (Uwah, A., & Umoren, I., 2024).

This article focuses specifically on the methodology for workbook management using MEOM within VBA (Chaudhry, A. K., Kalwar, M. A., 2021). Workbook operations form the foundation of any Excel-based application, whether it is a simple macro or a comprehensive business system. We will explore how to programmatically create, open, activate, modify, save, and protect workbooks, with practical code examples and explanations of key object properties and methods. Additionally, we will discuss the role of the Workbooks collection in managing multiple open files and highlight best practices for implementing security in automated environments.

By mastering these techniques, programmers and analysts can transform static spreadsheets into dynamic, secure, and intelligent systems, significantly enhancing productivity, ensuring data integrity, and reducing human error. This guide aims to provide both a conceptual understanding and a practical reference for implementing workbook-level automation, serving as a stepping stone toward more advanced Excel-VBA integration and application development (Ebere, F. O., Ekwueme, H., 2024).

Before examining the technical details of each workbook operation, it is essential to understand the fundamental workflow of programmatic workbook management in Excel VBA. The Excel Object Model (MEOM) provides a structured approach to workbook automation, which can be visualized as a sequential process:

Diagram 1. *Core Workbook Operations Sequence in Excel VBA*



This simplified flowchart represents the seven essential stages of workbook manipulation through MEOM:

1. **Create** – Initialize new workbooks using Workbooks. Add
2. **Open** – Load existing files with Workbooks.Open or Open Text
3. **Activate** – Set workbook context via Workbook. Activate
4. **Write** – Input data using Range. Value or Cells properties
5. **Save** – Persist changes through Save, SaveAs, or Save Copy As
6. **Protect** – Apply security with Protect or Protect Sharing
7. **Close** – Terminate sessions using Close (True/False)

While this linear representation shows the basic progression, actual implementations often involve loops, conditional branching, and error handling to accommodate real-world scenarios such as batch processing, template-based generation, and multi-user environments. The subsequent sections of this article will explore each operation in detail, providing practical code examples and implementation guidelines.

2. Creating a Microsoft Excel Workbook

While Microsoft Excel provides an intuitive graphical interface for manual workbook creation, advanced users and developers often need to generate workbooks programmatically. This capability is essential for automation, batch processing, and integration with other applications. Through Visual Basic for Applications (VBA) and the Excel Object Model (MEOM), workbook creation becomes a flexible and controllable operation.

The fundamental approach to programmatic workbook creation in Excel VBA is through the Add method of the Workbooks collection. This method belongs to the Application.Workbooks object, which represents all open workbooks in the current Excel instance.

The basic syntax is: Application.Workbooks. Add

This minimalist command creates a single new workbook based on Excel's default template. The workbook receives an automatically generated name following the pattern "Book N.xlsx" (where N is a sequential number, e.g., Book 1.xlsx, Book 2.xlsx). The new workbook immediately becomes

the active workbook in the Excel application window.

In many practical scenarios, creating a workbook is followed immediately by saving it to a specific location with a designated filename. This can be accomplished by chaining the SaveAs method to the creation command.

Example with immediate save: Application.Workbooks. Add. Save As "My Workbook. xlsx"

This compound statement performs two operations: creates a new workbook and saves it to the active directory as "My Workbook. xlsx".

The SaveAs method offers extensive parameterization for enhanced control including file format specification (XLSX, XLS, CSV, etc.), password protection for opening and/or modifying, read-only recommendations, and access mode settings for shared environments (Mertz, D., 2021).

Extended example with parameters:

Dim new Workbook As Workbook

Set new Workbook = Application. Workbooks. Add new Work book. Save As Filename:="Financial Report. xlsx", Password:="secure 123", File Format: = xl Open XML Workbook, Create Backup:=False

For standardized reporting, formatted dashboards, or predefined corporate templates, Excel supports creation from custom template files. Templates (with extensions. xltx,.xltm, or.xlt) preserve structure, formatting, formulas, and even VBA code (Alexander, M., & Walkenbach, J., 2013).

Template-based creation: Application. Work books. Add "Corporate Template. xltx"

This command generates a new workbook that inherits all elements from the specified template. The template file must be accessible from the current working directory or a fully qualified path must be provided.

Example with full path: Application. Workbooks. Add "C:\Templates\MonthlyReport.xltm" Advanced creation scenarios include creating multiple workbooks programmatically using loops: Dim i As Integer For i = 1 To 5 Dim wb As Workbook Set wb = Application.Work books. Add wb. Save As "Report_" & i & ".xlsx" wb. Close Next i Excel also provides constants for built-in template types. For example, Application. Workbooks.Add creates a workbook with a single worksheet (default), while Application.Workbooks. Add, 3 creates a workbook with three worksheets. It is important to properly handle the created workbook object for subsequent operations. The Add method returns a Workbook object reference that can be stored in a variable for later manipulation: Dim newBook As Workbook Set newBook = Application.Workbooks. Add Additional operations on newBook newBook.Worksheets (1).Range ("A1"). Value = "Report Title" newBook. Save As "Final Report. xlsx" The programmatic creation of workbooks represents the foundation of Excel automation, enabling developers to build sophisticated data processing systems, automated reporting tools, and integrated business applications within the familiar Excel environment.

3. Opening a Microsoft Excel Workbook

A workbook can be opened programmatically in two primary ways:

1. Using Excel's Built-in Dialog:

vba

Application.Dialogs.Item(xlDialogOpen). Show

This displays the standard Open File dialog to the user.

2. Using the Open **Method (Direct):**

vba

Application.Workbooks.Open Filename:="MyBook.xls"

This is the most straightforward method for opening a known file.

Excel can also import and convert structured text files. The OpenText method is used for this purpose. The following example opens a tilde-delimited text file:

```
vba
Workbooks.OpenText Filename:="101TB.txt", _
Origin:= –535, _
StartRow:=1, _
DataType:=xlDelimited, _
TextQualifier:=xlDoubleQuote, _
Other:=True, _
OtherChar:="~", _
FieldInfo:=Array(Array (1, 1), Array (2, 1), Array (3, 1), Array (4, 1)), _
TrailingMinusNumbers:=True
```

This code, which can be generated using Excel's Macro Recorder and then simplified,

creates a new workbook "101TB.txt.xls", places the data on a worksheet, and populates cells A1: D6 with the records from the text file.

3.1. Syntax of the MEOM Open Method

The Open method has a comprehensive set of optional parameters:

expression.Open(FileName, UpdateLinks, ReadOnly, Format, Password, WriteResPassword, IgnoreReadOnlyRecommended, Origin, Delimiter, Editable, Notify, Converter, AddToMru, Local, CorruptLoad)

CorruptLoad Parameter

Description:

This parameter determines how Excel should load a corrupted or damaged file**.** It controls Excel's behavior when a file reading error occurs during opening.

Values and Meanings:

Table 1.

Value	Constant Name	Description
1.	xlNormalLoad	Performs normal loading. Returns an error if the file is corrupted.
2.	xlRepairFile	Attempts to repair the file; if unsuccessful, extracts data from it.
3.	xlExtractData	Extracts only data from the file, ignoring formatting and other features.

Example Usage:

vba

' Open corrupted file with repair attempt:

Workbooks. Open File Name:="corrupted_file. xlsx", Corrupt Load:=xlRepair File

' Open with data extraction only:

Work books. Open File Name:="corrupted_file.xlsx", Corrupt Load:=xl ExtractData

When to Use:

- When suspecting file corruption
- In data recovery scenarios
- When accepting format loss to access critical data

AddToMru Parameter

Description:

AddToMru (Most Recently Used) controls whether the opened file should be added to Excel's "Recent Files" list.

Values:

- True: File is added to MRU list
- False: File is NOT added to MRU list (default)

Example Usage:

vba

' Open file and ADD to MRU list:

Workbooks. Open File Name:="report. xlsx", Add To Mru: = True

' Open file but DO NOT add to MRU list:

Workbooks. Open File Name: ="temp. xlsx", Add To Mru: =False

When to Use:

- Add To Mru: = False → Temporary files, automation-generated reports, hidden data files
- Add To Mru:= True → Frequently used templates, main data files, files requiring easy user access

Notify Parameter

Description:

This parameter controls Excel's user notification mechanism when a file is locked (being used by someone else or another program).

Values:

- True: If file is locked, it›s added to a «notification list.» Excel alerts the user when the file becomes available.
- False: Notification mechanism is disabled (default).

Example Usage:

vba

' Get notification if file is locked:

Workbooks. Open File Name:= " shared. xlsx", Notify: =True

' No notification required:

Work books. Open File Name:= " local. xlsx", Notify:= False

How It Works:

1. Attempt to open the file
2. If file is locked (opens in read-only mode)
3. If Notify:=True, file is added to "notification list"

4. When file becomes available, Excel shows a message in status bar or plays alert sound

When to Use:

- Notify:= True → Network shared files, multi-user environments
- Notify:=False → Local files, single-user environments, automation scripts

Comparison Table

Table 2.

Parameter	Default	Recommended Use	Performance Impact
Corrupt Load	xlNormal Load	When data recovery is needed	High (repair is resource-intensive)
Add To Mru	False	False for temp files, True for permanent files	Low
Notify	False	True for shared files, False for local files	Medium (background monitoring)

Expression – Required parameter. Returns a Workbooks object. This provides a reference to either the Workbooks collection or a single Workbook object.

File Name – Required String parameter. Specifies the complete file path of the file to be opened. Example: "C:\MyDocuments\report.xlsx".

Update Links – Optional parameter. Specifies how to update external links. Accepts values between 0 and 3. 0: do not update links, 1: update external links, 2: update remote links, 3: update both external and remote links.

Read Only – Optional Boolean parameter. Determines whether the workbook should be opened as read-only. A value of True opens the file in read-only mode.

Format – Optional parameter. Specifies the delimiter to be used for text files. Accepts values from 1 to 6: 1 (tab), 2 (comma), 3 (space), 4 (semicolon), 5 (none), 6 (custom character).

Password – Optional String parameter. Specifies the password required to open a password-protected workbook.

Write Res Password – Optional String parameter. Specifies the password required for write access. This password grants permission to modify and save the file.

Ignore Read Only Recommended – Optional Boolean parameter. Determines whether the "read-only recommended" prompt should be displayed if the file was saved with this setting. A value of True suppresses the warning.

Origin – Optional parameter. Specifies the platform origin of a text file. Common values include: xlMacintosh (for Macintosh), xlWindows (for Windows), and xlMSDOS (for MS-DOS). This parameter is important for correctly interpreting text encoding and line endings.

Delimiter – Optional String parameter. Specifies a custom delimiter character to be used when Format is set to 6 (custom). This allows importing text files with non-standard separators.

Editable – Optional Boolean parameter. Originally used for Excel 4.0 add-ins, this parameter is largely obsolete in modern Excel versions. It was designed to control whether add-ins should be opened as editable.

Notify – Optional Boolean parameter. Controls whether Excel should add a file to a notification list if it is locked by another user or application. When True, Excel monitors the file and notifies the user when it becomes available.

Converter – Optional parameter. Specifies the index of the file converter to use when opening files in non-native formats. This is useful for opening files created in other spreadsheet applications or legacy formats.

Add To Mru – Optional Boolean parameter. Determines whether the opened file should be added to Excel's Most Recently Used (MRU) file list. True adds the file to the list, False excludes it from the list.

Local – Optional Boolean parameter. Controls the language context for saving files. When True, files are saved in the language of Excel; when False, files are saved in the lan-

guage of VBA. This affects date formats, function names, and other locale-specific settings.

CorruptLoad – Optional parameter. Specifies how Excel should handle potentially corrupted files. Accepts three values: xlNormalLoad (normal loading with error if corrupted), xlRepairFile (attempt to repair the file), and xlExtractData (extract data only, ignoring formatting and formulas).

4. Activating a Microsoft Excel Workbook

When multiple workbooks are open, specific references are required. Activation can be done directly or by explicitly activating a workbook.

- Direct Reference Form:

vba

```
Workbooks("Kitab1.xls").Worksheets("Sehife1").Cells(1, 1) = "This book name is Kitab1.xls"
```

- Activation Form (Useful when switching active context):

vba

```
Application.Workbooks("Kitab1.xls").Activate
ActiveWorkbook.Worksheets("Sehife1").Cells(1, 1) = "This book name is Kitab1.xls"
```

After activation, the ActiveWorkbook object can be used for concise references. Navigation between workbooks must be managed carefully by the programmer.

5. Writing Information to a Microsoft Excel Workbook

Data can be written to cells by referencing the appropriate workbook, worksheet, and range. The following example demonstrates opening a password-protected workbook, writing data to specific cells, and closing it.

vba

```
Sub e_Open Book_With_Write Info()
Application. Workbooks. Open Filename:= "My Book. xls", Password:= "bla-bla"
With Workbooks("MyBook.xls"). Worksheets(1)
Cells(1, 1) = "This book name:"
Cells(1, 2) = "MyBook.xls"
Cells(2, 1) = "Loads count this book:"
Cells(2, 2) = 0 'Intended as a counter for future use
End With
Application.Workbooks("MyBook.xls"). Close (True)
End Sub
```

Line 02: Opens the workbook «MyBook. xls» using the password «bla-bla".

Line 03: Sets the target object to the first worksheet of the opened workbook.

Lines 04–07: Writes information into four cells. Cell (2,2) is initialized as a counter.

A subsequent procedure can modify this data:

vba

```
Sub e_Open Book_With_ChangeInfo()
Application. Workbooks. Open Filename:="My Book. xls", Password:= "bla-bla"
With Workbooks ("My Book. xls"). Worksheets (1)
.Cells(2, 2) =.Cells(2, 2) + 1 'Increments the counter
End With
Application.Workbooks("MyBook.xls"). Close (True)
End Sub
```

6. Saving a Microsoft Excel Workbook

MEOM provides methods corresponding to Excel's "Save" and "Save As" operations.

- Save **Method:** Saves the workbook to its known location.

vba

```
Application.Workbooks("MyBook1.xls"). Save
```

- Save Copy As **Method:** Saves a copy of the workbook without affecting the original open file.

vba

```
Application. Workbooks("My Book. xls"). SaveCopyAs ("Copy_MyBook_" & Int(Timer) & ".xls")
```

Example procedure:

vba

```
Sub e_Open Book_With_Save()
Application. Workbooks. Open Filename:= "My Book1. xls"
With Workbooks ("My Book1. xls"). Worksheets (1)
Cells(2, 2) =.Cells(2, 2) + 1
End With
Application.Workbooks("My Book1.xls"). Save
Application.Workbooks("My Book1.xls"). Close
End Sub
```

Note: The Close(True) method automatically saves changes, making an explicit Save call redundant in many cases, which programmers use to write more compact code.

7. Closing a Microsoft Excel Workbook

The Close method of the Workbook object is used. Its behavior depends on the parameter and the workbook›s state:

- Close(True)**:** Saves all changes and closes the workbook.
- Close(False)**:** If changes exist, Excel displays a dialog prompting the user to save. If no changes exist, it closes immediately.
- Close **(no parameter):** Behaves like Close(False).

In automated systems, the save/close logic is typically handled by the program (e.g., Close(True)) to avoid user interaction.

8. Working with the Workbooks Collection

The Workbooks collection is vital for managing multiple open workbooks within an application.

- *Listing All Open Workbooks***:**

vba

```
For Each obj In Workbooks
Debug. Print obj. Name
Next
```

- *Checking if a Specific Workbook is Open:*

vba

```
Sub e_Workbooks_CheckObject()
sFile = “My Book.xls”
bFind = False
For Each obj In Workbooks
If obj.Name = sFile Then
bFind = True
Exit For
End If
Next
If bFind Then
Msg Box (sFile & “ is loaded”)
Else
Msg Box (sFile & “ is not loaded”)
End If
End Sub
```

- *Saving and Closing All Open Workbooks:*

vba

```
For Each obj In Workbooks
obj.Close (True)
Next
```

9. Implementing Protection for a Microsoft Excel Workbook

MEOM allows for the programmatic control of Excel’s security features.

- Creating a Password-Protected Workbook:

vba

```
Application. Workbooks. Add. Save As Filename:=”My Book New. xls”, Password:=”12345”
```

- Opening a Password-Protected Workbook:

vba

```
Application. Workbooks. Open Filename:=”My Book. xls”, Password:= ”12345”
```

- Protecting Workbook Structure (Prevents adding/deleting/moving sheets):

vba

```
Application. Workbooks (“Other1.xls”). Activate
Active Workbook. Protect Password:= ”12345”, Structure:= True, Windows:= False
```

To unprotect:

vba

```
Active Workbook. Unprotect Password:= ”12345”
```

- Enabling Shared Workbook Protection (Network):

vba

```
Application.Workbooks(“Other2.xls”). Activate
Active Workbook. ProtectSharing Filename:= ”Other2.xls”, Sharing Password:= ”12345”
```

To disable sharing protection:

vba

```
ActiveWorkbook.UnprotectSharing SharingPassword:=”12345”
```

Conclusion

Programmatic control of Microsoft Excel through VBA and the Excel Object Model (MEOM) is a cornerstone of office automation and application development. This article provided a methodological overview of core workbook operations – creation, opening, activation, data manipulation, saving, closing, collection management, and protection. By leveraging these techniques, developers can transition from manual spreadsheet use to

building sophisticated, automated data processing systems that enhance productivity, ensure accuracy, and provide tailored business solutions within the familiar Excel environment.

The content presented here focuses primarily on foundational and intermediate-level workbook management, deliberately laying a solid groundwork for beginners and intermediate users. However, real-world Excel-VBA applications often require more advanced capabilities to create robust, maintainable, and responsive systems. Future work could extend this methodology to cover deeper topics such as:

- Event handling: Implementing workbook, worksheet, and application-level events (e.g., Workbook_Open, Worksheet_Change, BeforeSave) to build reactive and interactive applications that respond automatically to user actions or data changes;
- Custom classes and object-oriented design: Creating user-defined classes to encapsulate complex objects (e.g., custom data structures, reusable components) and fully exploit OOP principles within the VBA environment;
- Comprehensive error handling: Advanced techniques using On Error GoTo, Err object, custom error classes, and structured exception management to make automation scripts resilient against file corruption, user errors, network issues, or unexpected data conditions;
- Integration with external systems: Connecting Excel to databases (ADO/DAO), web APIs, or other Office applications for dynamic data import/export;
- Performance optimization and user forms: Efficient handling of large datasets, array-based operations, and development of professional user interfaces with custom forms and controls.

Exploring these advanced areas would enable the development of enterprise-grade Excel-based tools, further bridging the gap between simple automation and full-fledged application development. Researchers and practitioners are encouraged to build upon this foundation to address more complex automation challenges in business intelligence, scientific computing, and data analytics domains.

References

Alexander, M., & Walkenbach, J. (2013). Excel dashboards and reports.

Chaudhry, A. K., Kalwar, M. A., Khan, M. A., & Shaikh, S. A. (2021). Improving the Efficiency of Small Management Information System by Using VBA. International Journal of Science and Engineering Investigations, – 10(111). – P. 7–13.

Ebere, F. O., Ekwueme, H., Onwuama, C. N., Adu-Mensah, D., Nkok, L., & Josephs, R. (2024). Advancing Reservoir Performance Optimization through User-Friendly Excel VBA Software Development. Petroleum & Petrochemical Engineering Journal, – 8(1). – P. 1–20.

Evensen, H. T. (2014, June). A versatile platform for programming and data acquisition: Excel and Visual Basic for Applications. In 2014 ASEE Annual Conference & Exposition (P. 24–125).

Mertz, D. (2021). Cleaning data for effective data science. Packt Publishing.

Uwah, A., & Umoren, I. (2024). Soft Computing-Based System for Performance Modeling of Object-Oriented Programming (OOP) Software. European Journal of Computer Science and Information Technology, – 12(2). – P. 22–40.

submitted 07.12.2025;
accepted for publication 21.12.2025;
published 30.12.2025
© Bashirova G. I.
Contact: qoncabashirova@yahoo.com