UserForm1.Show

Posted on October 25, 2017March 28, 2018 by Rubberduck VBA

I've seen these tutorials. You've probably seen them too. They all go "see how easy it is?!" when they end with a glorious `UserForm1.Show` without explaining anything about what it means for your code and your understanding of programming concepts, to use a form's *default instance* like this. Most don't even venture into explaining anything about that *default instance* – and off you go, see you on Stack Overflow.

Because if you don't know what you're doing, all you've learned is how to write code that, in the name of "hey look it's so easy", abstracts away crucially important concepts that will, sooner or later, come back to bite you in the …*rear end*.

What's that *default instance* anyway?

A UserForm is essentially a class module with a designer and a `VB_PredeclaredId` attribute. That *PredeclaredId* means VBA is automatically creating a global-scope instance of the class, named after that class. If the default instance is ever unloaded or set to `Nothing`, its internal state gets reset, and automatically reinitialized as soon as the default instance is invoked again. You can `Set UserForm1 = Nothing` all you want, you can never verify whether `UserForm1 Is Nothing`, because that expression will always evaluate to `False`. A default instance is nice for, say, exposing a *factory method*. But please, *please* don't `Show` the default instance.

Doing. It. Wrong.™

There are a number of red flags invariably raised in many UserForm tutorials:

- `Unload Me`, or worse, `Unload UserForm1`, in the form's code-behind. The former makes the form instance a self-destructing object, the latter ~~destroys~~ *resets* the default instance, and that's *not necessarily* the executing instance – and that leads to all kinds of funky unexpected behavior, and embarrassing duplicate questions on Stack Overflow. Every day.
- `UserForm1.Show` at the call site, where `UserForm1` isn't a local variable but the "hey look it's free" *default instance*, which means you're using an object without even realizing it (at least without `New`-ing it up yourself) – and you're storing state that belongs to a *global instance*, which means you're using an object but without the benefits of object-oriented programming. It also means that…
- **The application logic is implemented in the form's code-behind.** In programming this [anti-]pattern has a name: *the "smart UI"*. If a dialog does *anything* beyond displaying and collecting data, it's doing someone else's job. That piece of logic is now *coupled* with the UI, and it's impossible to write a unit test for it. It also means you can't possibly reuse that form for something else in the same project (heck, or for something similar in another project) without making considerable changes to the form's code-behind. A form that's used in 20 places and runs the show for 20 functionalities, can't possibly be anything other than a spaghetti mess.

So that's what *not* to do. Flipside.

Doing it right.

What you want at the call site is to show *an instance of* the form, let the user do its thing, and when the dialog closes, the calling code pulls the data from the form's state. This means you can't afford a self-destructing form that wipes out its entire state before the [Ok] button's *Click* handler even returns.

Hide it, don't *Unload* it.

In .NET's *Windows Forms* UI framework (WinForms / the .NET successor of MSForms), a form's `Show` method is a function that returns a `DialogResult` enum value, a bit like a `MsgBox` does. Makes sense; that `Show` method tells its caller what the user meant to do with the form's state: `Ok` being your green light to process it, `Cancel` meaning the user chose not to proceed – and your program is supposed to act accordingly.

You see `Show`-ing a dialog isn't some *fire-and-forget* business: if the caller is going to be responsible for knowing what to do when the form is okayed or cancelled, then it's going to need to know whether the form is okayed or cancelled.

And a form can't tell its caller anything if clicking the [Ok] button nukes the form object.

The basic code-behind for a form with an [Ok] and a [Cancel] button could look like this:

```vb
Option Explicit
'@Folder("UI")
Private cancelled As Boolean

Public Property Get IsCancelled() As Boolean
    IsCancelled = cancelled
End Property

Private Sub OkButton_Click()
    Hide
End Sub

Private Sub CancelButton_Click()
    OnCancel
End Sub

Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    If CloseMode = VbQueryClose.vbFormControlMenu Then
        Cancel = True
        OnCancel
    End If
End Sub

Private Sub OnCancel()
    cancelled = True
    Hide
End Sub
```

Notice there are two ways to cancel the dialog: the [Cancel] button, and the [X] button, which would also nuke the object instance if `Cancel = True` wasn't specified in the `QueryClose` handler. Handling `QueryClose` is fundamental – not doing it means even if you're not `Unload`-ing it anywhere, [X]-ing out of the form will inevitably cause issues, because the calling code has all rights to not be expecting a self-destructing object – you need to have the form's object reference around, for the caller to be able to verify if the form was cancelled when `.Show` returns.

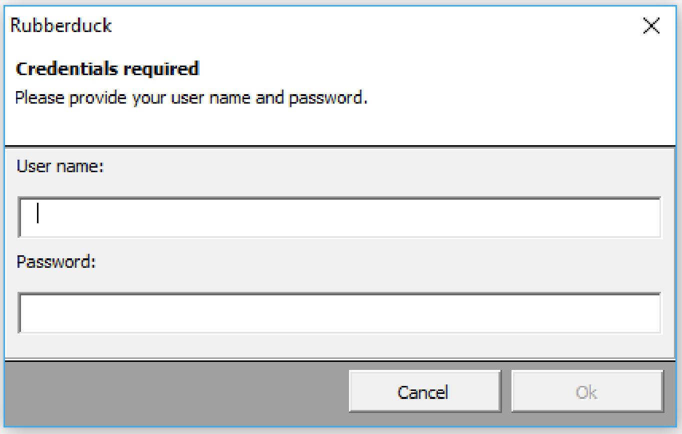
The calling code looks like this:

```vb
With New UserForm1
    .Show
    If Not .IsCancelled Then
        '...
    End If
End With
```

Notice there's no need to declare a local variable; the `With New` syntax yields the object reference to the `With` block, which properly destroys the object whenever the `With` block is exited – hence why `GoTo`-jumping out and then back into a `With` block is never a good idea; this can happen accidentally, with a `Resume` or `Resume Next` instruction in an error-handling subroutine.

The Model

A dialog displays and collects data. If the caller needs to know about a `UserName` and a `Password`, it doesn’t need to care about some `userNameBox` and `passwordBox` textbox controls: what it cares about, is the `UserName` and the `Password` that the user provided in these controls – the controls themselves, the ability to hide them, move them, resize them, change their font and border style, etc., is utterly irrelevant. The calling code doesn’t need *controls*, it needs a *model* that encapsulates the form’s *data*.


Rubberduck

Credentials required

Please provide your user name and password.

User name:

Password:

In its simplest form, the model can take the shape of a few `Property Get` members in the form’s code-behind:

```vba
Public Property Get UserName() As String
    UserName = userNameBox.Text
End Property

Public Property Get Password() As String
    Password = passwordBox.Text
End Property
```

Or better, it could be a full-fledged class, exposing `Property Get` and `Property Let` members for every property.

The calling code can now get the form’s data without needing to care about controls and knowing that the `UserName` was entered in a `TextBox` control, or knowing the `Password` without knowing that the `PasswordChar` for the `passwordBox` was set to *.

Except, it can – form controls are basically public instance fields on the form object: the caller can happily access them at will… and this makes the `UserName` and `Password` interesting properties kind of lost in a sea of MSForms boilerplate in *IntelliSense*. So you implement the *model* in its own class module instead, and use *composition* to encapsulate it:

```vba
Private viewModel As LoginDialogModel

Public Property Get Model() As LoginDialogModel
    Set Model = viewModel
End Property

Public Property Set Model(ByVal value As LoginDialogModel)
    Set viewModel = value
End Property
```

The model could be updated by the textboxes – it could even expose `Boolean` properties that can be used to enable/disable the [Ok] button, or show/hide a validation error icon:

```vba
Private Sub userNameBox_Change()
    viewModel.UserName = userNameBox.Text
    ValidateForm
End Sub

Private Sub passwordBox_Change()
    viewModel.Password = passwordBox.Text
    ValidateForm
End Sub

Private Sub ValidateForm()
    okButton.Enabled = viewModel.IsValidModel
    userNameValidationErrorIcon.Visible = viewModel.IsInvalidUserName
    passwordValidationErrorIcon.Visible = viewModel.IsInvalidPassword
End Sub
```

Now, a problem remains: the caller doesn’t *want* to see the form’ [illegible] ols.

The View

So we have a *model* abstraction that the *view* can consume, but we don't have an abstraction for the *view*. That should be simple enough – let's add a new class module and define a general-purpose `IView` interface:

```
Option Explicit
'@Folder("Abstractions")
'@Interface

Public Function ShowDialog(ByVal viewModel As Object) As Boolean
End Function
```

Now the form can *implement* that interface – and because the interface is exposing that `ShowDialog` method, we don't need a public `IsCancelled` property anymore. I'm introducing a `Private Type` at this point, because I like having only one private field:

```
Option Explicit
Implements IView
'@Folder("UI")

Private Type TView
    IsCancelled As Boolean
    Model As LoginDialogModel
End Type

Private this As TView

Private Sub OkButton_Click()
    Hide
End Sub

Private Sub CancelButton_Click()
    OnCancel
End Sub

Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    If CloseMode = VbQueryClose.vbFormControlMenu Then
        Cancel = True
        OnCancel
    End If
End Sub

Private Sub OnCancel()
    this.IsCancelled = True
    Hide
End Sub

Private Function IView_ShowDialog(ByVal viewModel As Object) As Boolean
    Set this.Model = viewModel
    Show
    IView_ShowDialog = Not cancelled
End Function
```

The interface can't be general-purpose if the `Model` property is of a type more specific than `Object`, but it doesn't matter: the code-behind gets *IntelliSense* and early-bound, compile-time validation of member calls against it because the `Private viewModel` field is an *implementation detail*, and this particular `IView` implementation *is* a "login dialog" with a `LoginDialogModel`; the interface doesn't need to know, only the implementation.

The [Ok] button will only ever be enabled if the model is valid – that's one less thing for the caller to worry about, and the logic addressing that concern is neatly encapsulated in the model class itself.

The calling code is supplying the model, so its type is known to the caller – in fact that `Property Get` member is just provided as a convenience, because it makes little sense to `Set` a property without being able to `Get` it later.

Speaking of the calling code, with the addition of a `Self` property to the model class (`Set Self = Me`), it could look like this now:

```
Public Sub Test()

    Dim view As IView
    Set view = New LoginForm

    With New LoginDialogModel

        If Not view.ShowDialog(.Self) Then Exit Sub

        'consume the model:
        Debug.Print .UserName, .Password

    End With 'model goes out of scope

End Sub 'view goes out of scope
```

If you read the previous article about writing unit-testable code, you're now realizing (if you haven't already) that this `IView` interface could be implemented by some `MockLoginDialog` class that implements `ShowDialog` by returning a test-configured value, and unit tests could be written against any code that consumes an `IView` rather than an actual `LoginForm`, so long as you've written it in such a way that it's the calling code that's responsible for knowing what specific `IView` implementation the code is going to be interacting with.

The model's validation logic could be unit-tested, too:

```
Const value As String = "1234"
With New LoginDialogModel
    .Password = value
    Assert.IsTrue(.IsInvalidPassword, "'" & value & "' should be invalid.")
End With
```

With a *Model* and a *View*, you're one step away from implementing the `New`-ing-up a *Presenter* class, an abstraction that completes the MVP pattern, a *much* more robust way to write UI-involving code than a *Smart UI* is.

Advertisements

AUTOMATTIC

<?php find_developers([
  'language' => PHP,
  'specialty' => SCALING,
  'location' => ANYWHERE,
]);

APPLY

REPORT THIS AD

AUTOMATTIC

We're hiring backend developers. Join us!

APPLY

REPORT THIS AD

Posted in OOP, rubberduck, tutorials, unit-testing, vbaTagged oop, tutorial, unit-testing, userforms, vba

Published by Rubberduck VBA

I'm Mathieu Guindon (Microsoft MVP Office Apps & Services, 2018), you may have known me as "Mat's Mug" on Stack Overflow and Code Review Stack Exchange. I manage the Rubberduck open-source project, whose goal is to bring the Visual Basic Editor (VBE) - VBA's IDE - into the 21st century, by providing features modern IDE's provide. *View all posts by Rubberduck VBA*

25 thoughts on “UserForm1.Show”

1. **ThunderFrame** *October 30, 2017* *Reply*

   The same principles can and should be used with Access Forms and Reports (which are also just class objects with slightly different designers, and default instances).

   Furthermore, showing a new instance of an Access Form or Access Report, opens the way for showing multiple instances of the same form or report, but with different models or with different parameters. That’s simply not possible when using a default instance, as you’d either need to toggle the state and UI, or you’d be trampling on state. The key thing to remember is that you need to keep a reference to each instance in scope – that can be achieved by using a collection variable to manage each of the form references. That allows, for example, for a user to drill into more than one record simultaneously, and view the results side-by side.

   1. **ThunderFrame** *October 30, 2017* *Reply*

      Of course, you can also show multiple instances of a VBA.UserForm, but only if you show the forms modeless – you just need to use a FormShowConstants.vbModeless parameter value with the Show method.

2. **jeffrey Weir** *November 7, 2017* *Reply*

   Damn…I feel like I’m only holding on by my fingernails to the wisdom encapsulated in this post (and the last), partly because being a self-taught programmer, I simply don’t understand some of the lingo. (e.g. Factory method?). It’s like I need to understand the stuff in this post in order to understand the stuff in this post. I think iit’s time to head back to my largely-glossed-over copy of Professional Excel Development to reread more carefully the stuff I glossed over when I last looked at it with the benefit of a couple of years more tinkering under my belt since I last opened it, before heading back here and hoovering up the wisdom. But back here I’m headed…I need to get better, and at least now this isn’t an unknown unknown 🙂

   Rubberduck VBA *November 7, 2017* *Reply*

   You’ll find information on factory methods in works on OOP design patterns.. I doubt an Excel read will mention it 😉

   Rubberduck VBA *November 7, 2017* *Reply*

   Ironically, it’s possible looking for ‘vba oop factory’ takes you right back here, to a short VBA/OOP series of articles I wrote a while ago!

   1. **jeffrey Weir** *November 9, 2017* *Reply*

      Cool, found your article. Will give it a read.

3. **johnallenrichter** *November 14, 2017* *Reply*

   I like your ambitions of improving VBA. I’m sure there is room for improvement but at the same time I must and have always said that VBA and MS Office Products are the greatest developing environment for business I have ever seen. And I’ve been writing programs since the DOS days of Fortran, Foxtrot, Clipper, dBase I, II, III, III+ and IV. MS created a genius platform and I’ll stand firmly behind you to improve it even further.

4. **CommonSense** *November 14, 2017* *Reply*

   Thanks for such a good article!

5. **Kostas K.** *November 18, 2017* *Reply*

Thanks, this is indeed a good article.

Quick question regarding the .Hide method.

Reference from MSDN:

Hide Method

"Hides an object but doesn't unload it.
When an object is hidden, it's removed from the screen and its Visible property is set to False. A hidden object's controls aren't accessible to the user, but they are available programmatically to the running application, to other processes that may be communicating with the application…"

https://msdn.microsoft.com/en-us/vba/language-reference-vba/articles/hide-method

Therefore am I correct in saying that every time we create a new instance of the form and Hide it, we release its reference but the object remains somewhere in memory until the application closes?

Public Sub Test()
Dim view As IView
Set view = New LoginForm

With New LoginDialogModel
'…
End With 'model goes out of scope
End Sub 'view goes out of scope

Rubberduck VBA *November 18, 2017* *Reply*

Hide merely makes the form invisible, it has nothing to do with memory management or object lifetime. An alternative could be to just explicitly set the Visible property to False =)

6. **Alex** *December 8, 2017* *Reply*

I've implemented this and it works. I still have a question though; how is it that when the view returns control back to the presenter class (calling code) the model state (in the scope of the with block) has been modified given that only a *copy* of the model was passed to the view as per the IView function definition; IView_ShowDialog(byVal viewModel as Object)?

Rubberduck VBA *December 8, 2017* *Reply*

You've been bitten by object-references-passed-ByVal 😉

When you pass an object reference by value, the method receives a copy *of the pointer* to the object. So when the view modifies property values on the model, it *is* the same object.

What the view *cannot* do, is change the pointer itself, because only a method that receives an object pointer ByRef would be allowed to do that.

1. **Alex Cook** *December 8, 2017*

ah ok; looks like i was bitten 🙂 so byVal passes a copy of the value on the stack, and if it's a pointer then it's a pointer; it doesn't dereference the pointer to the object

another question; I'm new to programming and I've heard of MVC and now, after reading this, MVP and even MVVM. I've read a bit on MVP and understand that MVP makes the presenter the middle man and that the M and the V shouldn't know about each other? But here we have the view encapsulating the model?

Rubberduck VBA *December 8, 2017*

You'll encounter MVC mostly in web projects. MVP is useful for MSForms/WinForms, and MVVM works nicely with WPF, because of the framework's almighty data binding capabilities. How each component relates to each other depends on what tutorial you're reading – from what I've read everyone has a different take on what the patterns should be, and varying degrees of religiousness in the implementation. In MVP IMO life is much easier when the View knows about and interacts with the Model (but not about the Presenter), the Presenter knows about the View and the Model, and the Model knows nothing but its own data. Your mileage may vary though. To make the View completely Model-agnostic you'd have to relay every single event to the Presenter, which makes the thing quite combersome I find.

7. **There is no worksheet. – Rubberduck News** *December 8, 2017 Reply*

[…] 0) – not counting anything test-related – and using this pattern (combined with MVP), the code is extremely clear and surprisingly simple; most macros look more or less like […]

8. **Userform closes after "End Sub" without ever calling "Unload Me" - QuestionFocus** *February 27, 2018 Reply*
   […] https://rubberduckvba.wordpress.com/2017/10/25/userform1-show/ […]

9. **Son Goku ssj4** *March 15, 2018* *Reply*

   Does anyone have working Workbook with this stuff written here? I'm lost in the sea of abstraction and would like to see this working before my eyes.

10. **elL** *May 8, 2018* *Reply*

    First of all, thanks a lot for this fantastic article. I'm re-implementing an old VBA Project (10k+ lines of code, sparsely documented, variables, fields, boxes. not named properly, etc.) and your articles have helped me keeping this whole project nice and clean.
    Now for the question, I would like to use a Userform to change the Value of certain objects (as in actual object in code), this works perfectly so far with the method above. But now there is the requirement for an "Apply Changes" Button on the UF, how would you handle this?
    The Idea is that the user can click this button and then something on the excel sheet gets updated, but without leaving the UF.
    The only solution I could come up with until now is using a Enum (or whatever) as return type of ShowDialog and then either use a GOTO to return back to the ShowDialog or use a Do…While Loop to repeat the ShowDialog every time Apply is clicked. But this doesn't strike me as particularly clean.

    Rubberduck VBA *May 8, 2018* *Reply*

    Hi, and thanks!

    That's a more involved scenario, you'll need a presenter class with the form as a private WithEvents field, and the macro/caller will then be working with that presenter object rather than directly with the form.

    What you want is to extract the 'Ok'/'Accept' logic into its own method (and still invoke it when the dialog is accepted), and then the 'Apply' button would raise an event (e.g. 'Public Event ApplyChanges()') in the form's ApplyButton_Click handler (without hiding the form) – then the code that invokes the form needs to be a class module (a "presenter" class) with a 'Private WithEvents dialog' field that holds a reference to the form being shown; the presenter can then handle 'dialog_ApplyChanges' by invoking the Ok/Accept logic that was pulled into its own method. Then the form might have its validation logic determine the form's "dirty" state and enable/disable the 'Apply' button accordingly.

    Not sure how clear that description can be in a comment, I should probably make another article to better illustrate. Let me know if I'm making any sense 😉

    1. **elL** *May 8, 2018*

       Thanks for the Input, I already managed to create a working Prototype of the Ok/Cancel/Apply UserForm and the Presenter. I'll extend it a bit post it if I think it is good enough. I actually managed the Reference to the Userform by attaching it to the Event.

       Below are the essential pieces of code:
       (is there a way to collapse/spoiler these answers?)

       OCAuserform Codebehind:

       '@Folder("VBAProject")
       Option Explicit
       Public Event ApplyChanges(Origin As OCAuserform, ByRef Cancel As Boolean)
       ……
       Private Sub ApplyChanges()
       Dim Cancel As Boolean: Cancel = False
       RaiseEvent ApplyChanges(Me, Cancel)
       If Cancel = False Then 'Non-Cancel Action
       Else 'Cancel Action
       End If
       End Sub

       OCApresenter:

```
‘@Folder(“VBAProject”)
Option Explicit
Private WithEvents aUF As OCAuserform

Public Sub ShowOCA()
Set aUF = New OCAuserform
With aUF
.tbValue = “Standart value”
.Show
End With
End Sub

Private Sub aUF_ApplyChanges(Origin As OCAuserform, Cancel As Boolean)
Debug.Print Origin.tbValue
End Sub

Calling Code:
Sub testOCA()
With New OCApresenter
.ShowOCA
End With
End Sub
```

Rubberduck VBA *May 8, 2018*

I’ve made a quick article with a download link, illustrating what I had in mind. Hope it helps!

11. **‘Apply’ logic for UserForm dialog – Rubberduck News** *May 8, 2018 Reply*
    […] recent comment on UserForm1.Show asked about how to extend that logic to a dialog that would have an “Apply” button. […]

12. **Cardin Pierre** *August 13, 2018* *Reply*

    Hi,

    I wanted to ask how to make your Userform stay on top/front? That no matter how many windows you open it will not go behind them.

    Thanks

    Rubberduck VBA *August 13, 2018* *Reply*

    The default is to show them modally; modal forms are on top of everything else, and must be closed before resuming. You can probably make a modeless form stick on top with some Win32 trickery, too. Depends what you’re trying to do specifically. FWIW a window that just sticks on top all the time isn’t normal window behavior in Windows.

13. **OOP Battleship Part 1: The Patterns – Rubberduck News** *August 27, 2018 Reply*
    […] instance. What’s critical is to avoid storing instance state in this default instance (see UserForm1.Show). But for pure functions such as factory methods, it’s […]