VBA Trap: Default Members

Posted on March 15, 2018June 6, 2018 by Rubberduck VBA

The key to writing clear, unambiguous code, is rather simple:

Do what you say; say what you do.

VBA has a number of features that make it easy to not even realize you're writing code that *doesn't do what it says it does*.

One of the reasons for that, is the existence of *default members* – under the guise of what appears to be *simpler code*, member calls are made *implicitly*.

If you know what's going on, you're probably fine. If you're learning, or you're just unfamiliar with the API you're using, there's a trap before your feet, and both run-time and compile-time errors waiting to happen.

Example

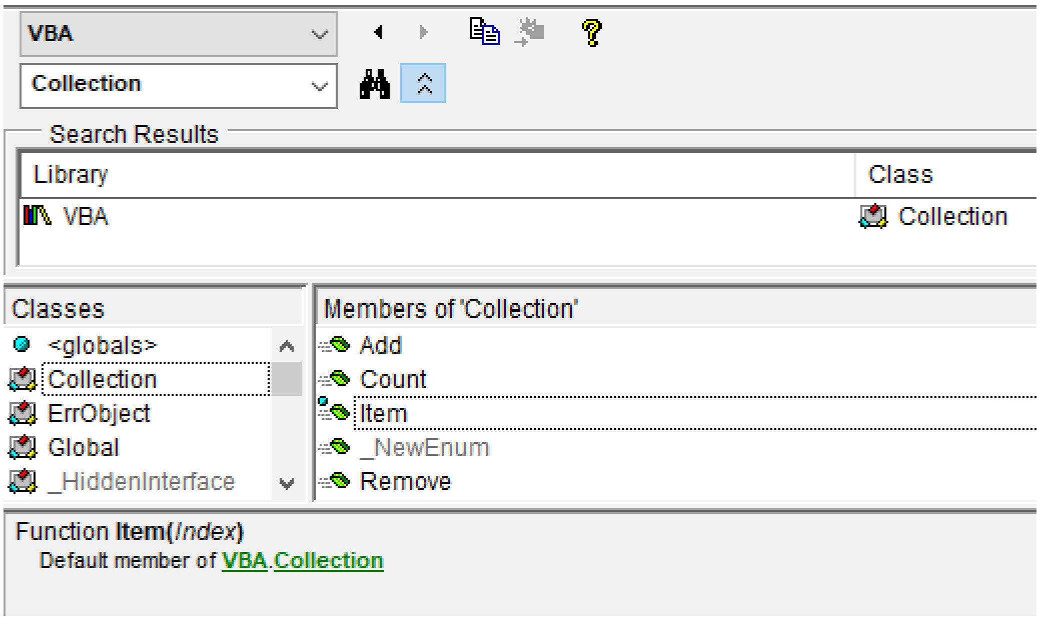
Consider this seemingly simple code:

```
myCollection.Add ActiveSheet.Cells(1, 1), ActiveSheet.Cells(1, 1)
```

It's adding a `Range` object, using the `String` representation of `Range.[_Default]` as a key. That's two **very** different things, done by two bits of **identical** code. Clearly that snippet does more than just what it claims to be doing.

Discovering Default Members

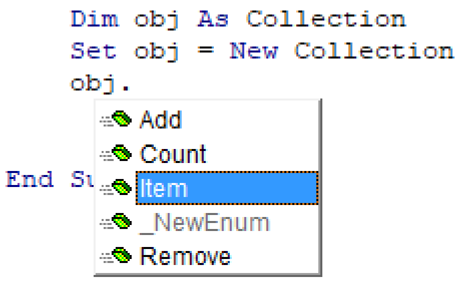
One of the first classes you might encounter, might be the **Collection** class. Bring up the *Object Browser* (F2) and find it in the **VBA** type library: you'll notice a little blue dot next to the **Item** function's icon:



Function **Item**(*Index*)
Default member of VBA.Collection

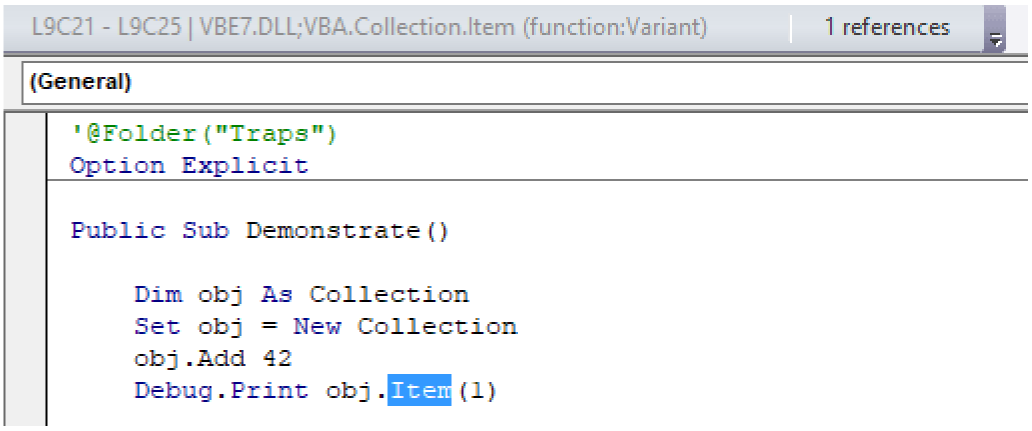
Whenever you encounter that blue dot in a of members, you've found the *default member* of the class you're looking at.

That's why the *Object Browser* is your friend – even though it can list hidden members (toggled via the *Object Browser*'s context menu), *IntelliSense* /autocomplete doesn't tell you as much:



```
    Dim obj As Collection
    Set obj = New Collection
    obj.
End Su
```

Rubberduck's context-sensitive toolbar has an opportunity to display that information, however that wouldn't help *discovering* default members:



L9C21 - L9C25 | VBE7.DLL;VBA.Collection.Item (function:Variant) | 1 references

(General)

```
'@Folder("Traps")
Option Explicit

Public Sub Demonstrate()

    Dim obj As Collection
    Set obj = New Collection
    obj.Add 42
    Debug.Print obj.Item(1)
```

Until Rubberduck reinvents VBA *IntelliSense*, the *Object Browser* is all you've got.

What's a *Default Member* anyway?

Any class can have a *default member*, and only one single member can be the default.

When a class has a *default member*, you can legally omit that member when working with an instance of that class.

In other words, `myCollection.Item(1)` is exactly the same as `myCollection(1)`, except the latter is *implicitly* invoking the `Item` function, while the former is explicit about it.

Can my classes have a *default member*?

You too can make your own classes have a default member, by specifying a `UserMemId` attribute value of `0` for that member.

Unfortunately only the `Description` attribute can be given a value (in the *Object Browser*, locate and right-click the member, select *properties*) without removing/exporting the module, editing the exported .cls file, and re-importing the class module into the VBA project.

An `Item` property that looks like this in the VBE:

```
Public Property Get Item(ByVal index As Long) As Variant
End Property
```

Might look like this once exported:

```
Public Property Get Item(ByVal index As Long) As Variant
Attribute Item.VB_Description = "Gets or sets the element at the specified i
Attribute Item.VB_UserMemId = 0
End Property
```

It's that `VB_UserMemId` member attribute that makes `Item` the default member of the class. The `VB_Description` member attribute determines the *docstring* that the *Object Browser* displays in its bottom panel, and that Rubberduck displays in its context-sensitive toolbar.

> **DANGER!**
> Rubberduck's module rewriters work off the code in the *code pane*, as it appears in the VBE. If Rubberduck makes a change (e.g. a refactoring, or an inspection quick-fix) in a class module that contains member attributes, **they will be lost**.
>
> This can cause compilation errors… if your code has implicit default member calls.

Whatever you do, don't make a default member that returns an instance of the class it's defined in. Unless you want to crash (https://stackoverflow.com/q/42075908/1188513) your host application as soon as the VBE tries to figure out what's going on.

What's Confusing About it?

There's an open issue (https://github.com/rubberduck-vba/Rubberduck/issues/3153) detailing the challenges implicit default members pose. If you're familiar with `Excel.Range`, you know how it's pretty much impossible to tell exactly what's going on when you invoke the `Cells` member (see Stack Overflow (https://stackoverflow.com/a/32997154/1188513)).

You may have encountered `MSForms.ReturnBoolean` before:

```
Private Sub ComboBox1_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
    If Not IsNumeric(Chr(KeyAscii)) Then KeyAscii = 0
End Sub
```

The reason you can assign `KeyAscii = 0` *and have any effect* with that assignment (noticed it's passed `ByVal`), is because `MSForms.ReturnInteger` is a class that has, you guessed it, a default member – compare with the equivalent explicit code:

```
Private Sub ComboBox1_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
    If Not IsNumeric(Chr(KeyAscii.Value)) Then KeyAscii.Value = 0
End Sub
```

And now everything makes better sense. Let's look at common Excel VBA code:

```
Dim foo As Range
foo = Range("B12") ' default member Let = default member Get / error 91
Set foo = Range("B12") ' sets the object reference '...
```

If `foo` is a `Range` object that is already assigned with a valid object reference, it assigns `foo.Value` with whatever `Range("B12").Value` returns. If `foo` happened to be `Nothing` at that point, run-time error 91 would be raised. If we added the `Set` keyword to the assignment, we would now be assigning the *actual* object reference itself. Wait, there's more.

```
Dim foo As Variant
Set foo = Range("B12") ' foo becomes Variant/Range
foo = Range("B12") ' Variant subtype is only known at run-time '...
```

If `foo` is a `Variant`, it assigns `Range("B12").Value` (given multiple cells e.g. `Range("A1:B12").Value`, `foo` becomes a 2D Variant array holding the values of every cell in the specified range), but if we add `Set` in front of the instruction, `foo` will happily hold a reference to the `Range` object itself. But what if `foo` has an explicit value type?

```
Dim foo As String
Set foo = Range("B12") ' object required
foo = Range("B12") ' default member Get and implicit type conversion '...
```

If `foo` is a `String` and the cell contains a `#VALUE!` error, a run-time error is raised because an error value can't be coerced into a `String` …or any other type, for that matter. Since `String` isn't an object type, sticking a `Set` in front of the assignment would give us an "object required" compile error.

Add to that, that `Range` is either a member of a global-scope object representing whichever worksheet is the `ActiveSheet` if the code is written in a standard module, or a member of the worksheet itself if the code is written in a worksheet module, and it becomes clear that this seemingly simple code is riddled with assumptions – and assumptions are usually nothing but bugs waiting to surface.

See, “simple” code really isn’t all that simple after all. Compare to a less naive / more defensive approach:

```vba
Dim foo As Variant foo = ActiveSheet.Range("B12").Value
If Not IsError(foo) Then
    Dim bar As String
    bar = CStr(foo) '...
End If
```

Now prepending a `Set` keyword to the `foo` assignment no longer makes any sense, since we *know* the intent is to get the `.Value` off the `ActiveSheet`. We’re reading the cell value into an explicit `Variant` and explicitly ensuring the Variant subtype isn’t `Variant/Error` before we go and explicitly convert the value into a `String`.

Write code that speaks for itself:

- Avoid implicit *default member* calls
- Avoid implicit global qualifiers (e.g. `[ActiveSheet.]Range`)
- Avoid implicit type conversions from `Variant` subtypes

Bang (!) Operator

When the default member is a collection class with a `String` indexer, VBA allows you to use the *Bang Operator* `!` to… implicitly access that indexer and completely obscure away the default member accesses:

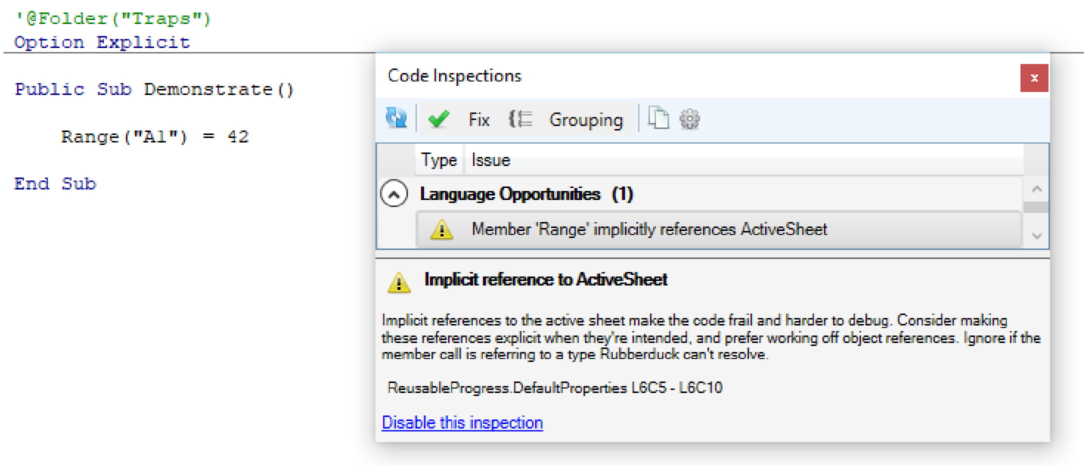
```vba
Debug.Print myRecordset.Fields.Item("Field1").Value 'explicit
Debug.Print myRecordset!Field1 'all-implicit
```

Here we’re looking at `ADODB.Recordset.Fields` being the default member of `ADODB.Recordset`; that’s a collection class with an indexer that can take a `String` representing the field name. And since `ADODB.Field` has a default property, that too can be eliminated, making it easy to… completely lose track of what’s really going on.

Can Rubberduck help / Can I help Rubberduck?

As of this writing, in theory Rubberduck has all the information it needs to issue inspection results as appropriate… assuming everything is early-bound (i.e. not written against `Variant` or `Object`, which means the types involved are only known to VBA at run-time).

In fact, there's already an Excel-specific inspection addressing *implicit ActiveSheet references*, that would fire a result given an unqualified `Range` (or `Cells`, `Rows`, `Columns`, or `Names`) member call.



```
'@Folder("Traps")
Option Explicit

Public Sub Demonstrate()

    Range("A1") = 42

End Sub
```

Code Inspections

Fix | Grouping

Type | Issue

Language Opportunities (1)

Member 'Range' implicitly references ActiveSheet

Implicit reference to ActiveSheet

Implicit references to the active sheet make the code frail and harder to debug. Consider making these references explicit when they're intended, and prefer working off object references. Ignore if the member call is referring to a type Rubberduck can't resolve.

ReusableProgress.DefaultProperties L6C5 - L6C10

Disable this inspection

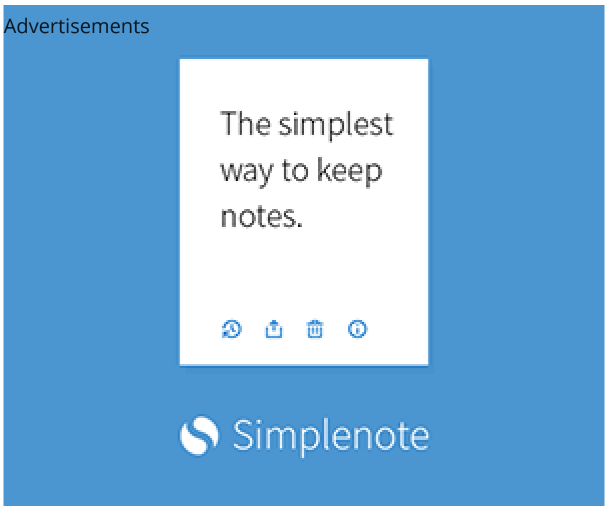
The inspection is currently firing a result even when the code is written in a worksheet module, making it a half-lie: without `Me.` qualifying the call, `Range("A1")` in a worksheet module is actually implicitly referring to *that worksheet*…and changing the code to explicitly refer to `ActiveSheet` would actually change the behavior of the code. That's actually a simple bug fix (https://github.com/rubberduck-vba/Rubberduck/issues/3569) that makes a good first issue for a first-time contributor! Are you this lucky person?

The reason it hasn't been fixed yet, is because *knowing* whether a given "document" module is a `Workbook` or a `Worksheet` instance, is a rather complex problem that has only been solved recently.

On the other hand, an inspection to flag implicit default member calls has yet to be implemented. That's a rather tricky one, because we need to actually *evaluate* the expressions involved, *resolve* them to a type, and determine if that type has a default member. Sounds easy? Take a stab at it (https://github.com/rubberduck-vba/Rubberduck/issues/2504)!

Let-assignments involving implicit type conversions are also something we need to look into. Help us do it (https://github.com/rubberduck-vba/Rubberduck/issues/2382)! This inspection also implies resolving the type of the RHS expression.

The reason these inspections haven't been implemented yet, is because there is essentially no expression-evaluation API in place; we need to leverage our existing resolver code and expose a nice entry point to use from within an inspection. If you're curious about Rubberduck's internals and/or would love to learn some serious C#, don't hesitate to create an issue (https://github.com/rubberduck-vba/Rubberduck/issues/new) on our repository to ask *anything* about our code base; our team is more than happy to guide new contributors in every area!



Posted in *open-source*, *rubberduck*, *tutorials*, *vba**Tagged* *attributes*, *object-browser*, *tutorial*, *vba*, *vbe*

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*

3 thoughts on “VBA Trap: Default Members”

1. **Felipe Costa Gualberto** *March 15, 2018* *Reply*

Another danger using default members:
Say that in a worksheet, A1 value is C3
Write "Hello" in cell C3.
In VBA, this works:
MsgBox Range(Range("A1").Value).Value
But this doesn't work:
MsgBox Range(Range("A1")).Value

Rubberduck VBA *March 15, 2018* *Reply*

I will be investigating that one. See I lied a little: the default member of Range isn't its Value property, it's a hidden [_Default] member that *appears* to ultimately resolve to Value… but who knows how it's implemented… What's the content of A1?

2. **Introducing the Object Browser – MyExcelMoments** *September 6, 2018 Reply*
[…] upper left corner as in or . Some classes designate a default members while others don't. https://rubberduckvba.wordpress.com/2018/03/15/vba-trap-default-members/ is a great post that explains […]