Clean VBA Code pt.2: Avoiding implicit code

Posted on October 25, 2018 by Rubberduck VBA

Clean code adheres to a number of principles. Does adhering to these principles make good code? Maybe, maybe not. But it definitely helps. One thing I find myself repeating quite a lot in my more recent Stack Overflow answers, is that code should "say what it does, *and do what it says*" – to me this means *writing explicit code*. Not just having `Option Explicit` specified, but avoiding the pitfalls of various "shortcuts" VBA lets us use to… cheat ourselves.

Avoid implicit member calls, write code that says what it does, and does what it says. Instead of:

```
1  Cells(i, 2) = 42
```

Prefer explicit qualifiers, and explicit member calls:

```
1  ActiveSheet.Cells(i, 2).Value = 42
```

In Excel, avoid working with **ActiveSheet** when you mean to work with **Sheet1**. Use the **Worksheets** collection instead of the **Sheets** collection when you mean to retrieve a *worksheet* in a workbook; *sheets* can contain charts and other non-worksheet sheet types.

```
1  Dim targetBook As Workbook
2  Set targetBook = Application.Workbooks.Open(path)
3
4  Dim targetSheet As Worksheet
5  Set targetSheet = targetBook.Worksheets("Sheet1")
6
7  Debug.Print targetSheet.Range("A1").Value
```

If the sheet we need exists in `ThisWorkbook` at compile-time, then we don't need a variable for it – it already exists:

```
1  Debug.Print Sheet1.Range("A1").Value
```

Every sheet in your Excel VBA project has a *code name* that you can set to any valid VBA module identifier name (up to 31 characters), and that identifier is now accessible from anywhere in your VBA project. To change the name, modify the `(Name)` property in the *properties* toolwindow (F4).

About the Bang! operator…

Avoid the Bang! operator. How many of the people using it know that the identifier to the right of the operator is a string literal that isn't compile-time validated? It *looks* like early-bound code, but it isn't. **The Bang! operator is an implicit default member call against a default member that takes a string parameter**. So this:

```
1 rs.Fields!Field1 = 42
```

Is really this:

```
1 rs.Fields.Item("Field1").Value = 42
```

Now, this doesn't mean we have to go crazy and dogmatic here – default properties *are* idiomatic, and not necessarily toxic… when used *carefully*. The `Item` member of a collection class is, by convention, the default member of the class:

```
1 rs.Fields("Field1").Value = 42
```

Note that `Fields` is plural, which strongly signals that `("Field1")` is an *indexed property accessor* (it is)… and we could even infer that it returns a `Field` object reference. There's an implicit default member call happening, yes, but it's pure syntax sugar here: even if we don't know that `Fields` is a class with a default `Item` property, we can tell that syntactically, we're invoking something, getting an object reference back and assigning its `Value` property with a value.

Contrast with `rs.Fields!Field1 = 42`, which reads like… witchcraft, come to think of it.

As an Excel programmer I'm biased though: Access programmers probably see the *Bang!* operator differently. After all, it's *everywhere*, in *every tutorial* – why would it suddenly be *wrong*?

Pros:

- Faster to type (?).
- Encourages using standard PascalCase field names and collection keys. Kinda.

Cons:

- Confusing syntax for an unfamiliar reader; makes a string look like a member access. That one's arguably on the reader/maintainer to read up, yes. Still.
- No compile-time validation: what follows the `!` operator is a *string*. Option Explicit will not save you from a typo.
- If any explicit member call follows the string, it is inherently late-bound and not compile-time validated either; the editor will offer no *intellisense* for it.
- Requires otherwise rather uncommon `[square bracket]` tokens around the name when the name contains spaces.

You have to put the Bang! operator in context: 25 years ago, using fully spelled-out variable names was seen as wasteful and borderline ludicrous. Code was written to be *executed*, not *read*: the faster you could type, the better. Oh, how things have changed!

Here's a screenshot from an old, deleted Stack Overflow question about the Bang! operator in…
VB.NET:

That construct looks ugly IMO. It feels like it makes string literals appear like keywords... – [redacted] Dec 4 '09 at 8:49

I remember using this operator in VB Classic (well, Access VBA). I had no idea it had survived into VB.NET! – [redacted] Dec 4 '09 at 11:42

4 please forget that it has survived... – [redacted] Dec 4 '09 at 11:53

1 OMG, this still exists in .Net? Now I have a second means to confuse my colleagues, the first being starting comments with REM. – [redacted] Dec 4 '09 at 16:04

My eyes are burning – [redacted] Sep 1 '11 at 10:48

The Bang! operator is a relic of the past. There's no reason to use it in modern code, be it in VBA, VB6… or VB.NET.

Posted in rubberduck, tutorials, vbaTagged bang-operator, best-practices, default-members, implicit, 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*