Out of Control Modularity

Jeremy Bante

https://www.linkedin.com/in/jbante/
https://github.com/jbante

```
Enter Find Mode [ Pause: Off ]

# For each find request...
Set Variable [ $request.i ; Value: 0 ]
Loop [ Flush: Always ]
    Set Variable [ $request ;
    Value: JSONGetElement ( Get ( ScriptParameter ) ; "requests[" & $request.i & "]" ) ]
    Exit Loop If [ IsEmpty ( $request ) ]
    If [ $request.i > 0 ]
        New Record/Request
    End If

    # For each request criterion (field-value pair)...
    Set Variable [ $criterion.i ; Value: 0 ]
    Loop [ Flush: Always ]
        Set Variable [ $criterion ;
        Value: JSONGetElement ( $request ; "criteria[" & $criterion.i & "]" ) ]
        Exit Loop If [ IsEmpty ( $criterion ) ]
        Set Variable [ $field ; Value: JSONGetElement ( $criterion ; "field" ) ]
        Set Variable [ $value ; Value: JSONGetElement ( $criterion ; "value" ) ]
        Set Field By Name [ $field ; $value ]
        Set Variable [ $criterion.i ; Value: $criterion.i + 1 ]
    End Loop

    Set Variable [ $omit ; Value: JSONGetElement ( $request ; "omit" ) ]
    If [ $omit ]
        Omit Record
    End If

    Set Variable [ $request.i ; Value: $request.i + 1 ]
End Loop

Set Variable [ $type ; Value: JSONGetElement ( Get ( ScriptParameter ) ; "type" ) ]
If [ $type = "find" ]
    Perform Find [ ]
Else If [ $type = "extend" ]
    Extend Found Set [ ]
Else If [ $type = "constrain" ]
    Constrain Found Set [ ]
End If

Exit Script [ Text Result: "" ]
```

I was reviewing some code when I came across this modular find script. On its face, it seems like a perfectly reasonable, highly modular script. It can handle any find I want. It has no dependencies to worry about. I can copy & paste it into any FileMaker file and it will work fine.

I have a problem with this script, it’s not just that there’s no error handling, and I want to explain why. But I have to get a little abstract first…

Abstraction

…starting with the word "abstraction". Programmers love to talk about how "abstract" code is or isn’t, and show off how "powerful" their "abstractions" are. Please stop.

~~Abstraction~~

- Generalization
- Encapsulation

The word "abstraction" means different things to different people in different contexts on different days of the week. It’s confusing because it isn’t clear which meaning we’re talking about at any given moment, making programming more difficult than it needs to be. Most of the time, we mean to talk about one of two completely separate concepts: generalization and encapsulation.

Consider a French press. A French press can be used to make coffee using many different sources of ground coffee beans: it generalizes over different grinds.

French press image: By KoeppiK - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=94493780

Coffee cup image: By Julius Schorzman - Own work, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=107645

Generalization

General code works in many different conditions

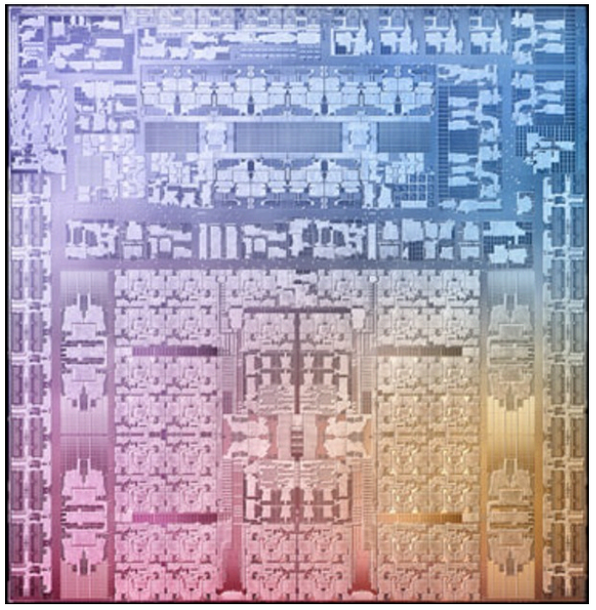
- One script or custom function…
  - Different applications
  - Different layouts
  - Different parameters
  - etc.

Consider a coffee machine. Given the inputs of ground coffee and water, a coffee machine can produce coffee via what may as well be magic while we're blissfully unaware of how the coffee is being made: it encapsulates the process of making coffee.

Encapsulation

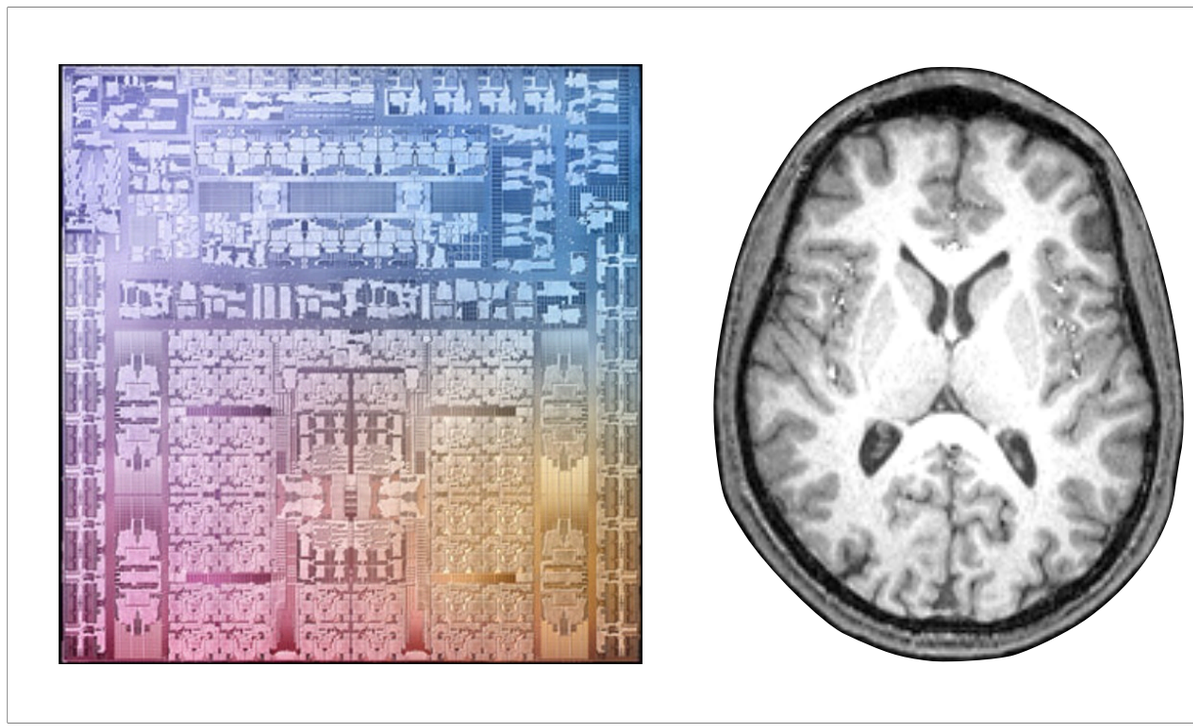
Encapsulated code can be used without thinking about how it works

- Chunks of logic that can be considered separately
- Details hidden while thinking about the bigger picture



Consider a computer. It can feel powerful when we write highly general code. But when we write highly general code, we don’t somehow imbue the computer with a capability it didn’t already have. Computers are already general-purpose machines. Programming actually does the opposite: we constrain the vast potential of what the computer could do down to the relatively few things we actually want it to do. Generalizing code is in essence about constraining the computer less.

For all their vast potential, computers are made of profoundly stupid components. It’s mind-boggling how many machine instructions it takes to perform even simple operations like setting a field with a fixed value. Programming such unwieldy contraptions is a challenge, and the crux of the issue is us.



Our human meat-brains have severely limited working memory—not nearly enough to think about those machine instructions all at once. Good computer programming is as much about human psychology as it is about computers.

This is where encapsulation comes in. With encapsulation, hundreds or thousands of machine instructions get wrapped up in a single Set Field script step. I don’t have to know what those machine instructions are or understand how they amount to setting a field; I just have to know how to use the script step.

Generalization < Encapsulation

So generalization can be a nice-to-have labor savings or productivity multiplier, but encapsulation is what makes it possible to program modern applications at all. Programmers aspiring to modularity and the Don't-Repeate-Yourself (DRY) principle emphasize the value of generalization, but encapsulation is ultimately more important.

With that in mind, let's revisit the modular find script…

```
Enter Find Mode [ Pause: Off ]

# For each find request...
Set Variable [ $request.i ; Value: 0 ]
Loop [ Flush: Always ]
    Set Variable [ $request ;
    Value: JSONGetElement ( Get ( ScriptParameter ) ; "requests[" & $request.i & "]" ) ]
    Exit Loop If [ IsEmpty ( $request ) ]
    If [ $request.i > 0 ]
        New Record/Request
    End If

    # For each request criterion (field-value pair)...
    Set Variable [ $criterion.i ; Value: 0 ]
    Loop [ Flush: Always ]
        Set Variable [ $criterion ;
        Value: JSONGetElement ( $request ; "criteria[" & $criterion.i & "]" ) ]
        Exit Loop If [ IsEmpty ( $criterion ) ]
        Set Variable [ $field ; Value: JSONGetElement ( $criterion ; "field" ) ]
        Set Variable [ $value ; Value: JSONGetElement ( $criterion ; "value" ) ]
        Set Field By Name [ $field ; $value ]
        Set Variable [ $criterion.i ; Value: $criterion.i + 1 ]
    End Loop

    Set Variable [ $omit ; Value: JSONGetElement ( $request ; "omit" ) ]
    If [ $omit ]
        Omit Record
    End If

    Set Variable [ $request.i ; Value: $request.i + 1 ]
End Loop

Set Variable [ $type ; Value: JSONGetElement ( Get ( ScriptParameter ) ; "type" ) ]
If [ $type = "find" ]
    Perform Find [ ]
Else If [ $type = "extend" ]
    Extend Found Set [ ]
Else If [ $type = "constrain" ]
    Constrain Found Set [ ]
End If

Exit Script [ Text Result: "" ]
```

Someone probably thought this was a well-abstracted script. We now see that this is a very general script. It works in any context, on any fields, to perform any find, in any file. Since we want modular scripts to be portable across applications, this seems to be desirable.

But let’s look at what it’s like to use this script.

```
Enter Find Mode [ Pause: Off ]
Set Field [ Contact::nameFirst ; "a" ]
Set Field [ Contact::company ; "Apple" ]
New Record/Request
Set Field [ Contact::nameFirst ; "*e*" ]
Set Field [ Contact::company ; "Apple" ]
Omit Record
Perform Find [ ]
```

```
Perform Script [ Specified: From list ; “Perform Specified Find” ; Parameter: JS
```

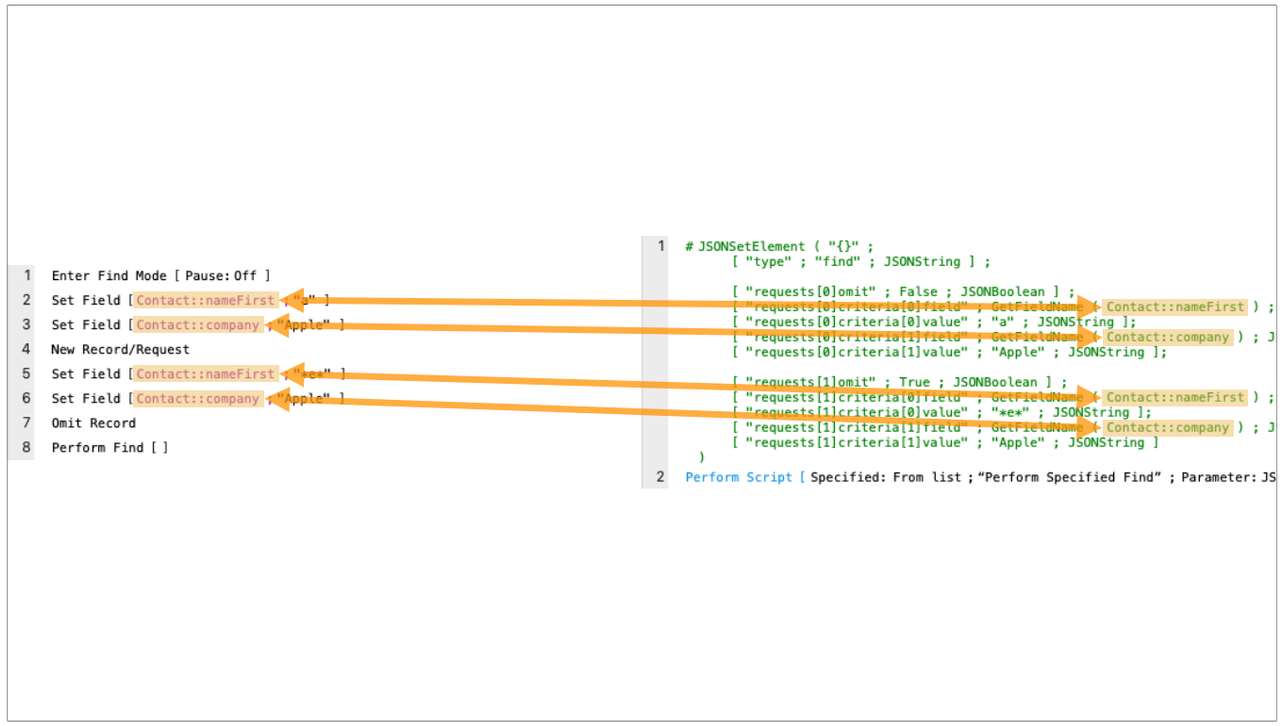
On the right, we’re performing a simple find. On the left, we’re performing the same find using our modular script. It looks like we’ve encapsulated 8 script steps down to 1. This is an illusion created by the way FileMaker formats script steps on-screen. Each step is displayed on 1 line, regardless of how much configuration goes into it.

Our find script requires a parameter, and we are just as responsible for setting that parameter on the right as we are for configuring each step on the left.

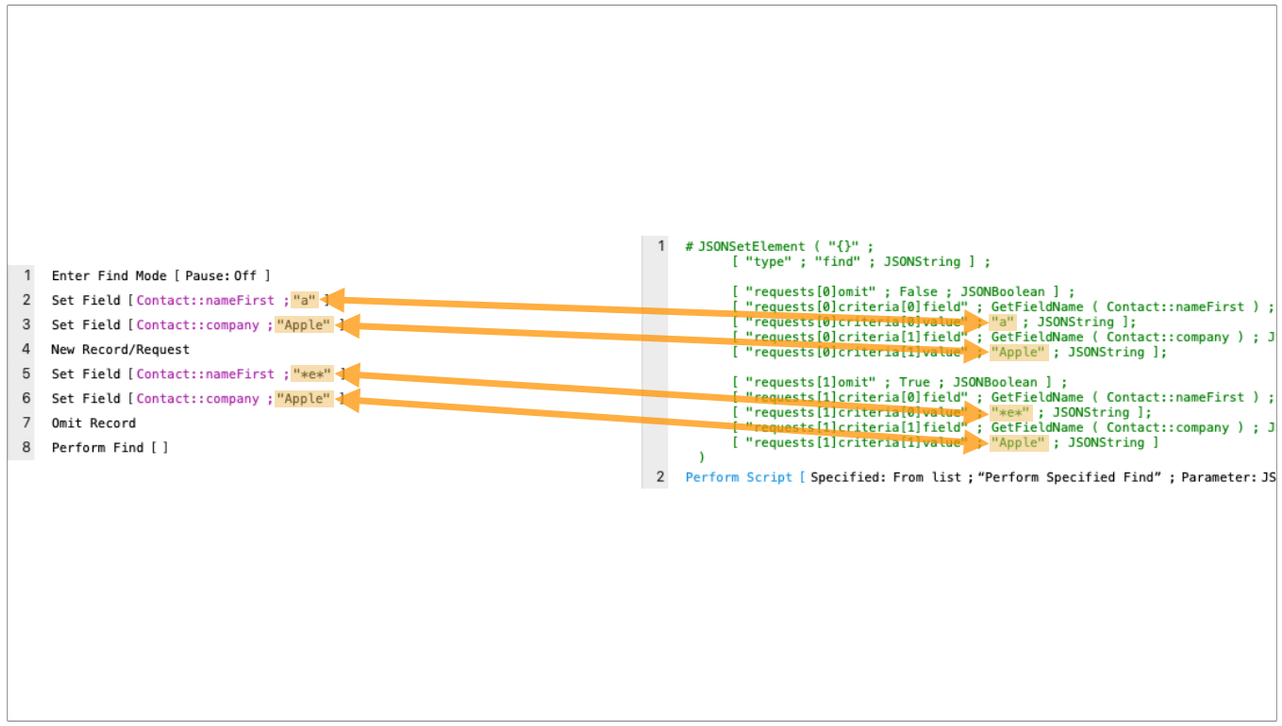
```
Enter Find Mode [ Pause: Off ]
Set Field [ Contact::nameFirst ; "a" ]
Set Field [ Contact::company ; "Apple" ]
New Record/Request
Set Field [ Contact::nameFirst ; "**" ]
Set Field [ Contact::company ; "Apple" ]
Omit Record
Perform Find [ ]
```

```
# JSONSetElement ( "{}" ;
      [ "type" ; "find" ; JSONString ] ;

      [ "requests[0]omit" ; False ; JSONBoolean ] ;
      [ "requests[0]criteria[0]field" ; GetFieldName ( Contact::nameFirst ) ;
      [ "requests[0]criteria[0]value" ; "a" ; JSONString ];
      [ "requests[0]criteria[1]field" ; GetFieldName ( Contact::company ) ; J
      [ "requests[0]criteria[1]value" ; "Apple" ; JSONString ];

      [ "requests[1]omit" ; True ; JSONBoolean ] ;
      [ "requests[1]criteria[0]field" ; GetFieldName ( Contact::nameFirst ) ;
      [ "requests[1]criteria[0]value" ; "**" ; JSONString ];
      [ "requests[1]criteria[1]field" ; GetFieldName ( Contact::company ) ; J
      [ "requests[1]criteria[1]value" ; "Apple" ; JSONString ]
 )
Perform Script [ Specified: From list ; “Perform Specified Find” ; Parameter: JS
```

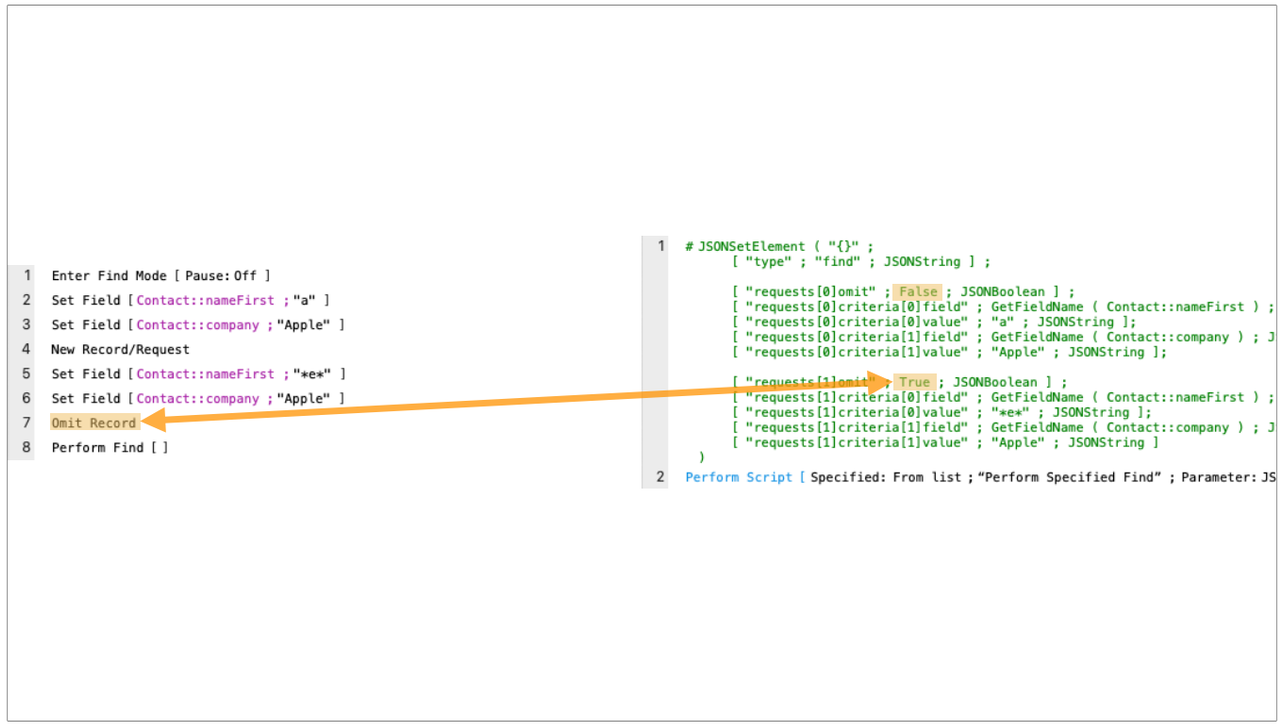
The parameter reveals what it’s really like to use this script: our modular find script is actually every bit as complicated to use as writing out the find for ourselves.



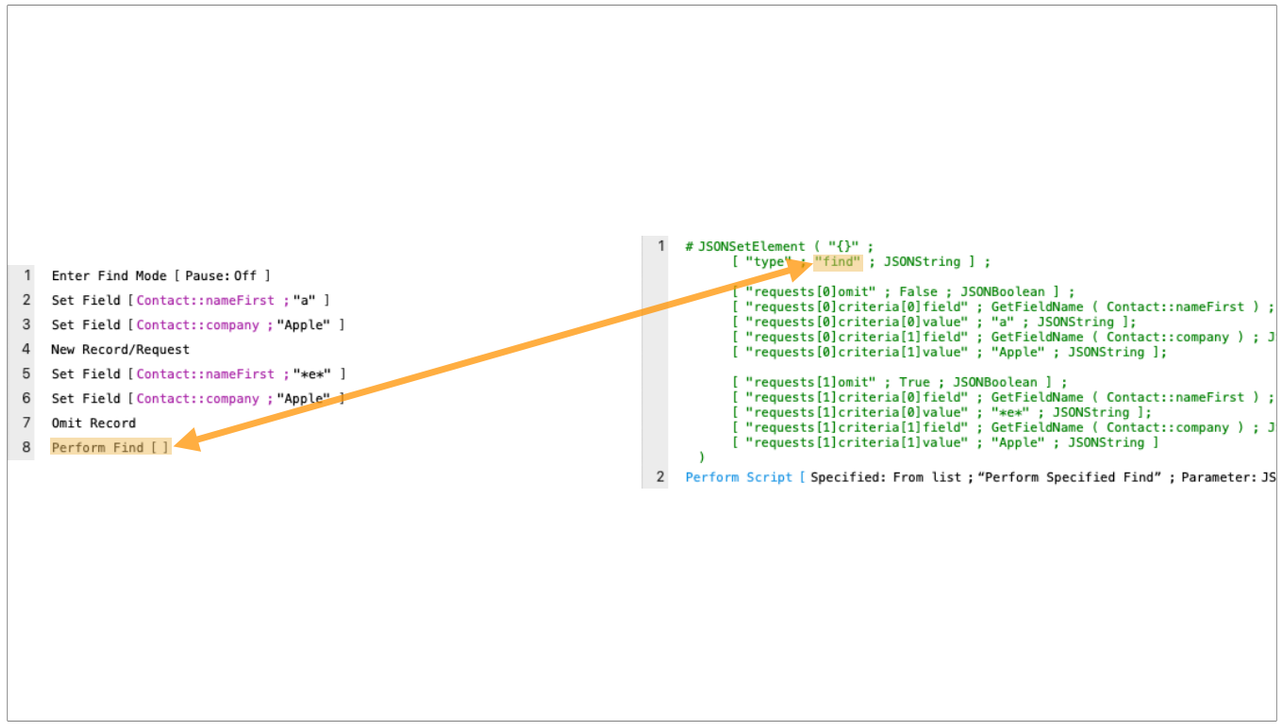
We still have to specify what fields to set.



We still have to specify our match criteria for each field.



We still have to specify which requests are for including vs. omitting records.



We still have to specify whether we’re performing a fresh find, or doing an extend or constrain. If anything, the script parameter is more complicated than just scripting the find for ourselves—this script encapsulates nothing, and that offends my sensibilities as a professional developer.

This is to say nothing of the extra work the computer has to do.

```
Enter Find Mode [ Pause: Off ]
Set Field [ Contact::nameFirst ; "a" ]
Set Field [ Contact::company ; "Apple" ]
New Record/Request
Set Field [ Contact::nameFirst ; "~~e~~" ]
Set Field [ Contact::company ; "Apple" ]
Omit Record
Perform Find [ ]
```

```
Enter Find Mode [ Pause: Off ]

# For each find request...
Set Variable [ $request.i ; Value: 0 ]
Loop [ Flush: Always ]
    Set Variable [ $request ;
    Value: JSONGetElement ( Get ( ScriptParameter ) ; "requests[" & $request.i & "]" ) ]
    Exit Loop If [ IsEmpty ( $request ) ]
    If [ $request.i > 0 ]
        New Record/Request
    End If

    # For each request criterion (field-value pair)...
    Set Variable [ $criterion.i ; Value: 0 ]
    Loop [ Flush: Always ]
        Set Variable [ $criterion ;
        Value: JSONGetElement ( $request ; "criteria[" & $criterion.i & "]" ) ]
        Exit Loop If [ IsEmpty ( $criterion ) ]
        Set Variable [ $field ; Value: JSONGetElement ( $criterion ; "field" ) ]
        Set Variable [ $value ; Value: JSONGetElement ( $criterion ; "value" ) ]
        Set Field By Name [ $field ; $value ]
        Set Variable [ $criterion.i ; Value: $criterion.i + 1 ]
    End Loop

    Set Variable [ $omit ; Value: JSONGetElement ( $request ; "omit" ) ]
    If [ $omit ]
        Omit Record
    End If

    Set Variable [ $request.i ; Value: $request.i + 1 ]
End Loop

Set Variable [ $type ; Value: JSONGetElement ( Get ( ScriptParameter ) ; "type" ) ]
If [ $type = "find" ]
    Perform Find [ ]
Else If [ $type = "extend" ]
    Extend Found Set [ ]
Else If [ $type = "constrain" ]
    Constrain Found Set [ ]
End If

Exit Script [ Text Result: "" ]
```

There’s a complicated script parameter to encode and parse out. There’s nested looping. There are checks for options we may or may not be using in any particular find.

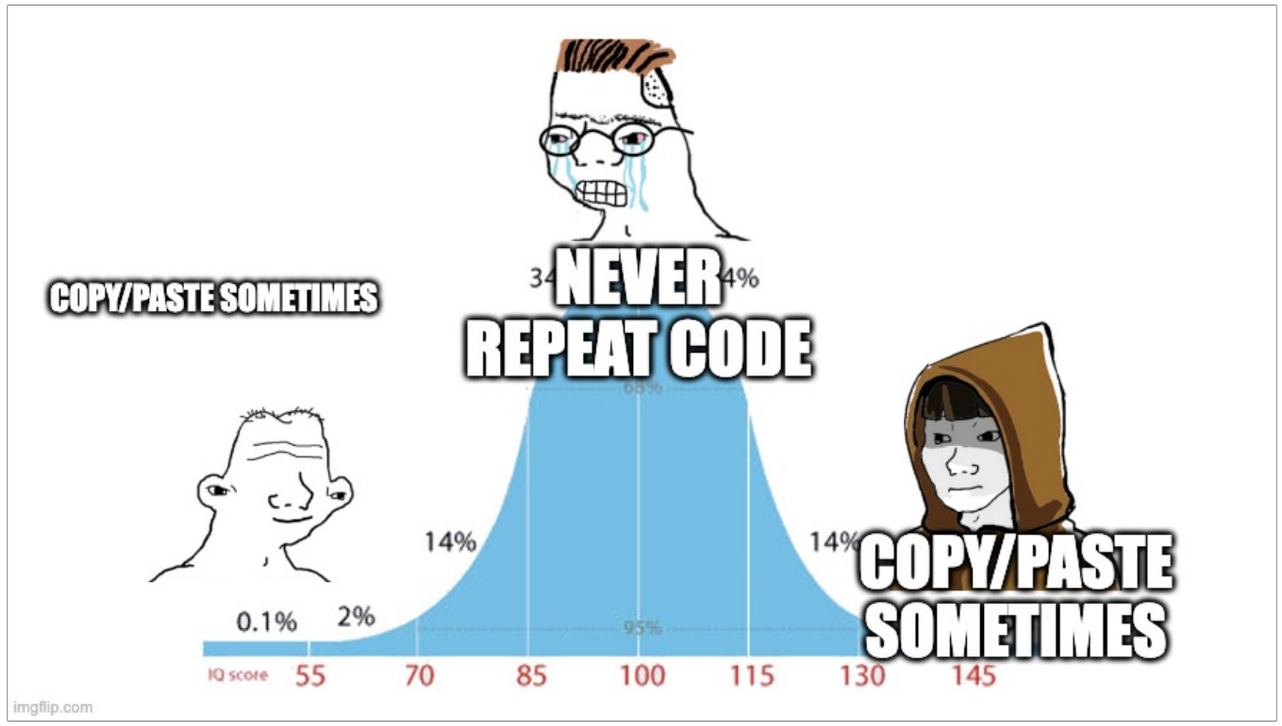
And all of this would be perfectly acceptable if the script actually encapsulated something for us, but it doesn’t. This script uses extra CPU cycles, and for that we get no psychological benefit or labor savings.

All code should encapsulate something

- Do you have to think about how a script or custom function works to use it?
- Do you have fewer parameters to specify compared to copy & paste?
- Does your script make some decision for you?

All code should encapsulate something.
• Do you have to think about how a script works in order to use it?
• Do you have fewer parameters to specify than if you recreated the same functionality for yourself?
• Is it less work to use your modular script than to copy, paste, and edit an equivalent block of code?
• Does your script make some decision on your behalf so you can think about other things?

And if not…



It’s perfectly OK to copy, paste, and edit blocks of code if there’s certain pattern you want to repeat. Not everything benefits from being packaged in a reusable module.

Image: Cory House: https://x.com/housecor/status/1796257950509728132