Programmer’s Guide to Mini-OOF

Introduction

Mini-OOF is a small library for ESP32forth that enables the programmer to write object oriented code. This library was originally created by Bernd Paysan specifically with microcontrollers in mind. It packs quite a punch for it’s size.

Why is object oriented code useful? Well – forth programming is all about creating a problem-specific language with which to construct an application. Programming in basic forth is all very well – but it’s like driving in 1st gear all the time. Better to create more sophisticated components that allow you to write your program in ‘top gear’ with *less* writing and *better* readability. Definitely worth a honk on the horn.

Mini-OOF isn’t a panacea for all applications – compare it with solutions in plain forth and choose the best one based on your own personal goals.

Extending the Compiler

The traditional tools for creating new classes of components are the words CREATE and DOES>. These are fine for simple to medium things, but code readability rapidly goes down as the component complexity rises. We’ve all heard the complaint that Forth is a write-only language. This issue is illustrated by comparing two very similar components:-

- This code for creating as many new stacks as you like. It’s very readable, very few variables are needed (1!) per stack. That variable is stored as the first cell in each stack, followed by the actual stack block.
- This code is for a very similar component - a queue. Per queue, we need the variables *head, tail, size, used* + the data block. Again, the variables are nearest to the start of the definition (although that’s hard to tell), with the data block following. Notice several things – the location and size of each variable isn’t obvious; and we need quite a few ‘helper’ words to access them – cluttering up the dictionary with low-grade special-to-queues words. Overall, understanding how the queue works is not as easy as the code for the stack, despite being closely related.

Both pieces of code are reasonably well commented – but that doesn’t improve readability that much. Hand-made object orientation works but is hard to maintain. Here’s Dick Pountain’s book about that kind of code.

If we go onto create more sophisticated components, like a recipe for creating state-machines for instance, then readability can dip even lower and accidentally introducing bugs gets more likely.

Enter Object Oriented Programming

So – we probably want to create components (Objects) for the following reasons:-

- We may have a need to create a fixed number of named components which all work the same way. They all have their own private variables as part of the internal machinery and they all share code – they all use the same 'Methods' to function. The components could be as simple as a stack or as complex as an Actor.
- We may want to create a variable number of temporary, unnamed components or Objects during program operation to react to conditions as the application runs. These Objects may need to be destroyed later when no longer useful, so they don't take up memory any more. (e.g. catering for a varying number of users connecting and disconnecting to an application via WiFi)
- We may need to create Objects that are very similar to other Objects, with one or two different or extra behaviours. How to do that with the smallest amount of writing / in the smallest code space without tying ourselves in knots?

Writing a recipe for making components – the Class

A Class defines the variables (internal storage) and methods (code that makes the Class do it's thing) as a self contained recipe. In Mini-OOF this looks like:-

Creating a Class

A CLASS is the recipe for making as many identical components as we need. The components created from the CLASS's recipe are called Objects. Each Object has internal variables (called VARs) and code (called METHODs) to may operate on the internal variables, data on the stack, other variables and values in memory etc.

```
OBJECT CLASS                    \ This Class's parent is Object, a 'do nothing' ancestor
        4 VAR variable1name     \ This defines a variable 4 bytes in size
        2 VAR variable2name     \ This defines a 2 byte variable
        METHOD method1name      \ This is a method that will run code
        METHOD method2name      \ As many as required for this Class
END-CLASS ClassName1            \ Classname1 is your name for this Class
```

So we're defining a Class called 'ClassName1' with internal data (the two VARs, one is a long, the other is a word) and code to make it all work (the two Methods). Notice that the Methods don't do anything yet – they're just place-holders for now.

Also, compare this with CREATE … DOES> : The anonymous variables are now clearly named and sized and a Class can execute any number of Methods whereas CREATE DOES> executes one 'Method' only. Result - much more powerful, more readable component-maker.

Defining the Methods

Our Class won't do anything unless we define what the Methods actually do. Here's the syntax for that:-

```
:NONAME   <your forth code for the method>   ;    ClassName1   DEFINES   method1name
```

An anonymous word is linked to 'method1name' in 'ClassName1' by the DEFINES word.

That seems like a long-winded way of defining a Method – defining it's name and then defining the code separately – why? Well - suppose within the definition of 'method1name' we wanted to call 'method2name'. If these two methods were ordinary forth words, we would be making a call to code that had yet to be compiled; but forth doesn't allow forward references. In the Class definition above all the Method names pre-exist, so that any Method may call any other Method to get the job done – no forward references involved.

The same syntax is used to define all the Methods for 'ClassName1' and we're done. 'ClassName1' is a little factory waiting to churn out as many identical gizmos as we like, each with their own private variable block.

Creating a Named Object

Here's the code to create a named Object, which actually does stuff. We name an object when we know it's going to be needed throughout the life of the application:-

```
ClassName1   NEW   CONSTANT   MyObjectName1
```

What's happened here? ClassName1 is the 'recipe' for making the new Object. NEW sets up the data space needed for all the internal VARs. MyObjectName1 is a constant to store the new Object's identity (ID) – simple as that. If we want more then …

```
ClassName1   NEW   CONSTANT   MyObjectName2
ClassName1   NEW   CONSTANT   MyObjectName3
```

… does the job, like a cooky cutter making biscuits.

Running Objects

If we want to access an Object's VARs, then this is not too different from using Forth's ordinary variables:-

```
MyObjectName1          variable1name   @                  \ read long 'variable1name'
10   MyObjectName1   variable2name   W!                 \ set word 'variable2name' = 10
```

Whereas we're used to a single word for a variable, VARs must be preceded with the name of the required Object. These little code fragments can be run from the terminal when debugging a Class, as well as included in a word definition - very handy.

The same applies to running Methods:-

```
MyObjectName1   method1name                \ run method1name
MyObjectName1   method2name                \ run method2name
```

Notice again that **the Object name has to be on the top of the data stack** before calling the method, so that the right bunch of VARs will be used – if the method uses any of them. This requirement affects the code inside a method ...

Calling other Methods and Vars from inside a Method

As we saw in the last section – when calling a Method or VAR it is always preceded by the required Object ID – so the Object ID is top of stack when a method starts to execute. Mini-OOF *expects that to be consumed* before the end of the method:-

```
:NONAME <your forth code for the method> ; ClassName1 DEFINES method1name
                ↑ the Object ID is top of          ↑ before  ;  the method must consume the
                   the data stack here                 Object ID by executing DROP or equivalent
```

Having the Object ID top of data stack is both convenient and inconvenient. We can DUP it before calling other Methods or VARs of the same Class – convenient. BUT, if we want the method to read input parameters or leave output parameters, then the Object ID on top of the data stack is in the way – inconvenient. The most obvious thing to do is to move the Object ID to the R stack as shown here:-

```
:NONAME
>R                                      \ transfer the Object ID to the R stack
< forth code>
R@ variable1name @ 1+
R@ variable1name !                      \ increment variable1name, get Object ID from R stack
< more forth code >
R>  method2name                         \ call  method2name and drop Object ID
;  ClassName1 DEFINES method2name
```

This is how a Method uses other Methods and VARs from its own Class. The Object ID is still handy on the top of the R stack, but doesn't mask any i/o parameters or intermediate results on the data stack.

Let's look at some real code – taken from this code for Timers. You see the use of >R to save the Object ID at the start, repeated use of R@, then finishing with an R> to consume the Object ID before executing the ; word to finish the TRUN method definition:-

```
:noname >R
MS-TICKS DUP                                  \ read the present time
R@ STARTTIME @                                \ read when this TIMER last ran
-                                             \ calculate how long ago that is
R@ PERIOD @ >=                                \ is it time to run the TCODE yet?
IF
   R@ STARTTIME !                             \ save the present time
   R> TCODE @ EXECUTE                         \ run xt stored in TCODE
ELSE
   DROP  R> DROP                              \ else forget the present time
THEN ; TIMER DEFINES TRUN                     \ run TCODE every PERIOD ms
```

Creating an Unnamed Object

We create unnamed Objects when (a) they are temporary in nature or (b) there would be too many Named Objects to handle elegantly.

You're building a machine which manages up to 20 streams of data, the number of channels required is held in value **chan#**. You could create 20 named Objects to deal with that, but calling the names wouldn't be too handy when it came to running each channel's code – quite a lot of writing. We could do with handling the channels as an array of Object IDs, we don't care about naming them. We also don't want to waste memory on unused channels.

The syntax for creating an array of unnamed objects might be:-

```
CREATE Channel 20 allot                       \ this array will store Object IDs or 0 if inactive
20 0 DO 0 Channel I cells +   !  LOOP          \ initialise all Channel[I] = 0
chan# 0 DO
ClassName1    HNEW    Channel I cells +   !    \ Channel[I] = Object ID
LOOP
```

Notice the use of **HNEW** instead of NEW. HNEW allocates space on the heap to store all the internal VARS needed by Classname1 Objects, instead of using program space.

Now, when processing each channel, we can access VARs or execute Methods within a DO LOOP with the phrase:-

```
Channel I cells + @  <methodname> or <VARname>
```

If a channel becomes inactive, then we don't want to process that channel any more and can remove that channels' object from memory with:-

```
Channel I cells +   @   FREE                  \ Return the channel VAR space to the heap
0 Channel I cells +  !                        \ Mark the channel as inactive
```

We can use the heap word FREE to discard the memory space used by the Object:-

```
chan# 0 DO
  Channel I cells +                          \  Index into the array
  DUP @   0<>   IF                           \  If the channel is active
        DUP @ FREE                           \  Return Object data to the heap
        0  SWAP    !                         \  and mark the channel inactive
  THEN
LOOP
```

We have now destroyed all the channel Objects we had created with HNEW – the heap size is as it was before any of the Objects were created.

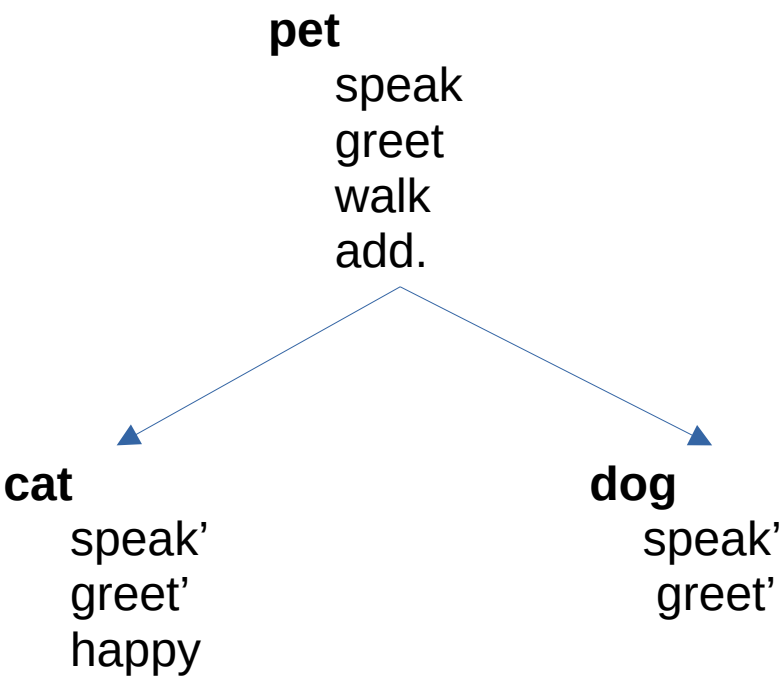
Creating New Classes similar to existing ones

Suppose we wish to create a new Class which is very similar to one we've already created? We *could* start from scratch – but would most likely produce a lot of duplicated code, which is wasteful. Also keeping any code in sync when a change was made is also a pain and bugs would creep in. Instead we can use Inheritance to derive a new Class from an existing one. The new Class inherits all the VARs and Methods from the old Class – and we just define the differences of the new Class.

This turns out to be more useful than you might think – if you're a newcomer to object oriented programming like me. An example is:-

You want to manage a list of shape Objects that need drawing on the display. All shapes are 'related', in that they need to be drawn, moved, erased and so on. Being able to cycle through the list applying a 'draw' Method to each shape in turn is a very powerful technique. We let Mini-OOF determine which 'draw' Method is selected for each Object ID so that lines, circles, rectangles and so on are drawn correctly. I wouldn't want to do that in normal forth – it would mean a large case statement to maintain (and expand when new shapes were added) - very easy to break.

Some very simple test code I wrote when I was converting Mini-OOF to ESP32forth shows how to make related Classes via inheritance:-



Above is the required Class hierarchy:-

A base Class 'pet' has four methods 'speak','greet', 'walk' and 'add.'
We create two Classes based on 'pet' called 'cat' and 'dog'. Left like that, we would find that Mini-OOF had made sure that both cats and dogs could do the same methods as pets. However, because cats and dogs do behave slightly differently, we modify their 'speak' and 'greet' methods. Additionally, we decide that cats can do extra stuff – so we define an extra method called 'happy'. Yes – I know dogs can be happy too, but humour me for now.

Let's look at the code - we define the pet Class:-

```
object class
        cell var teeth#
        cell var height
        method speak
        method greet
        method walk
        method add.
end-class pet
:noname ." pet speaks" drop  ; pet defines speak
:noname ." pet greets" drop   ; pet defines greet
:noname ." pet walks" drop   ; pet defines walk
:noname  drop + ." n1 + n2 = " . ; pet defines add.   ( n1 n2 -- )
```

It would be a good idea to make a pet Object and test out the Methods to make sure they work OK before moving on – but we decide to press on.

We create a cat Class:-

```
pet class
        method  happy           \ cats can do more than pets
end-class cat
:noname ." cat purrs" drop ; cat defines happy
\ cats override pets for these two methods
:noname ." cat says meow" drop ; cat defines speak
:noname ." cat raises tail" drop ; cat defines greet
```

- To ensure that cat inherits all the behaviour of pet, we declare '**pet class**' at the top, not 'object class'
- You can see defining a new Method for '**happy**' involves no new syntax
- Neither does redefining '**speak**' and '**greet**' – just make sure the same names are used as in 'pet'.
- We could have added more VARs, they're declared just like normal
- If a Method needs i/o parameters on the data stack, make sure all pet Classes use the same 'interface'

Now we create the dog Class:-

```
pet class
end-class dog
\ dogs override pets for these two methods
:noname ." dog says wuff" drop ; dog defines speak
:noname ." dog wags tail" drop ; dog defines greet
```

Nothing new there.

It's definitely time to test this all out by running some small code phrases from the terminal. To prepare for that, let's create named cat and dog Objects:-

```
\ create a cat and dog object to work with
cat new constant tibby
dog new constant fido
```

Let's test – can our cat and dog walk ok? – after all, that is defined in the pet Class:-

```
--> tibby walk
pet walks ok
--> fido walk
pet walks ok
```

Yes, so this shows that the inheritance of Methods from the parent Class works fine.

What about ‘speak’ and ‘greet’?

```
--> tibby greet
cat raises tail ok
--> fido greet
dog wags tail ok
--> tibby speak
cat says meow ok
--> fido speak
dog says wuff ok
--> tibby happy
cat purrs ok
```

Here we see that both methods have been overridden and when the animals execute those, they do so in different ways. Also notice that the ‘happy’ method we added to cat works. If you try using ‘fido happy’ the system will crash – fido doesn’t know how to do ‘happy’ – sorry dog lovers.
One last test – both animals should be able to do some simple maths – these are bright pets:-

```
--> 34 56 fido add.
n1 + n2 = 90  ok
--> 12 34 tibby add.
n1 + n2 = 46  ok
```

Yes, this method inherited and unchanged from ‘pet’ works well. Notice the method takes in two parameters from the data stack to do the sum. It could equally well leave results on the stack if that was required. Notice the definition of ‘add.’ drops the Object ID out of the way, so the parameters can be accessed:-

```
:noname  drop + ." n1 + n2 = " . ;    pet    defines    add.        ( n1 n2 -- )
```

The add. Method could have used >R, R@ and R> to preserve the Object ID as we’ve already covered - had there been a need to read and write from the Classes’ VARs or to call other Methods.

In summary, creating new Classes based on parent Classes - ‘Inheritance’ has the following benefits:-

- It economises on compiled code – it encourages factoring which is one of the goals of writing good forth code
- It means we write less source code and that’s good for readability too
- Allows us to code for related objects in a more general fashion e.g. We don’t care what kind of pet we’re dealing with, we can be confident that they’ll all speak
- When we add a new pet type – say donkey – we just create the new Class, we don’t have to fix any other locations in our code.

Calling Methods revisited – early and late binding

We've already seen how to call a Method:-

```
<ObjectName> <Methodname>
```

When these two words run, they compute the address of the code to execute and call it. That takes a certain amount of extra time, but it is useful when a variety of related Objects need to be processed by the Method, like we discussed in the previous section. This is termed 'late binding' because it happens at run-time.

However, if we're sure that the Object will always be the same, we can use 'early binding' – the address of the Method to run is calculated during compilation and the call to the Method is placed in-line in the definition. The syntax for that is:-

```
[   <ObjectName>   ::    <Methodname> ]
```

This code fragment must only be used inside definitions – forth will crash if you try it out from the terminal. The advantage in doing this – the new word being compiled is smaller and runs a bit faster because the address calculation is already done for us.

Here's two examples of doing the same thing to make the syntax a bit clearer:-

```
: TEST1 TIMER1 [ TIMER :: TPRINT ] CR ;                    \ Early binding
: TEST2 TIMER1 TPRINT CR ;                                  \ Late binding
```

However if we want a Method to work with many Objects:-

```
: TEST2 TPRINT CR ;                ( Obj – )                    \ Late binding
```

Only late binding will do – notice this word expects the Object on top of the stack – we assume all 'Obj' can TPRINT, else our application will come tumbling down in a big crash!

Crash protection (health warning)

As you may have found out in trying Mini-OOF code, it is all too easy to cause forth to crash. Mini-OOF is tiny but (as far as I can tell) stable if there are no mistakes in your code. However it offers no protection from using the wrong Methods or VARs with an Object. Trying to execute a Class when you meant to use an Object is similarly fatal. In these respects it's no different from abusing any other Forth word. We all know how to use the reset button from long experience!

Initialising Objects

As it stands, if we want Objects to be initialised, then that is up to us to code. However, we can arrange that when an Object is created it will also initialise. Let us define a new Class which will self-initialise:-

```
 INITOBJECT CLASS
          cell VAR myvar
END-CLASS MyInitClass
:noname myvar ! ;    MyInitClass    DEFINES    INIT
```

We've based our class on a special class defined in Mini-OOF – **INITOBJECT** and we've redefined it's **INIT** method to suit our purposes – to initialise myvar with a value off the stack. When we come to create some Objects it would be like this:-

```
80  MyInitClass    NEW:   MyObject1
55  MyInitClass    NEW:   MyObject2            \ and so on
```

Let's confirm initialisation has worked:-

```
--> MyObject1 myvar @ .
80  ok
--> MyObject2 myvar @ .
55  ok
```

So myvar took on the required starting value in each Object.

Conclusion

Hopefully this has been enough to wet your appetite for programming with Mini-OOF on ESP2forth; and you find like me that it makes some jobs a lot easier. Don't be tempted to turn everything into Objects – it's just one more tool in the box to use when appropriate.

It's worth looking at the rest of my Forth stuff too.

This paper was written by Bob Edwards, retired EMC engineer and ham radio operator G4BBY in S.W. U.K. in September 2022.

Appendix 1 Internal Stuctures

Mini-OOF constructs a couple of small tables, one when a Class is created, another when an Object is created. If you extend Mini-OOF, then knowing the makeup of these tables is useful:-

Object →

OBJECT table
VTABLE pointer
VAR1
VAR2
VAR3
etc..

→

VTABLE
Total space in bytes needed for OBJECT table
Total space in bytes needed for VTABLE
xt of METHOD1
xt of METHOD2
xt of METHOD3
etc.

← Class

(All table entries are 1 cell in size)

If an Object is created using NEW or NEW: , then the Object table is placed in code space. If the Object is created using HNEW, then the Object table is created in the Heap.

When a Class is run, it places the address of it's VTABLE on the data stack. This table contains the following entries:-

- Total space in bytes needed for OBJECT table – so if the Class has three long Vars, then this cell would contain 4 + 3 *4 = 16
- Total space in bytes needed for the VTABLE – if the Class has three Methods, then this address would contain 2 * 4 +3 * 4 = 20
- A list of cells, each containing a pointer to the code start address (xt) of a Method in the order in which they were defined in the Class.

When an Object is run, it places the address of the Object table on the stack as the 'Object ID'. This table contains the following entries:-

- VTABLE pointer – this cell contains a pointer to the Class's VTABLE, so that the Object has access to it's methods etc.
- The VARs are then stored in the following cells in the order in which they were defined.

When a VAR is run, then it uses a stored offset to calculate the start address of the VAR within the OBJECT table. The VAR start address is put on the data stack.

When a Method is run, it uses VTABLE pointer to find the VTABLE start, then adds an internal offset to point to the Method's xt. The xt is then read and executed. (xt – start address of a forth word's code)