Introduction

File OOP.FS holds the definition for an OO-mechanism, which loads on top of gforth 0.62. It makes maximum use of the existing Forth mechanisms to realize OO and therefore, it is very concise - 330 lines of code realize all mechanisms, which can be expected from an OO package to the best of my knowledge. (ARRAYs have not been included, but they are easy to add).

At present the code is dependent on gforth 0.62, because of the way the interpreter and compiler has to be redefined. It could be easily ported to gforth 0.71 as well. I believe that the concepts used can be expressed in Standard Forth, but I am not an expert in it and therefore, I would need to collaborate with one if needed.

File OBJECTS.FS defines classes CELL, BUFFER, and STRING and a couple of methods as a proof of concept. These definitions are followed by some test code, which predominantely test proxy handling and late binding.

This work is based on the ideas put forward by Manfred Mahlow for the first time. His concepts revolved around the so called "prelude concept", which would simplify class context setting without the need to make object defining words immediate and state smart. But using the prelude concept requires a radical modification of forth's headers, which requires system dependent modifications and recompilation of the Forth system.

Initially I implemented the OO package in the cross-compiler for uCore - without late binding. But Classes, Objects, Attributes, Proxys, and Arrays and their inheritable methods were already there. Thanks to Andrew Haley's insistence I started to think about an efficient late-binding mechanism. Especially, after I realized that late binding comes in productively for embedded applications as well, if you want to tweak code in a running system that has a complex state, as e.g. machine control. Which for me had been a hacking job without agreed upon structures. It dawned on me that late binding is the socially correct way of code patching.

The solution I found is embarrassingly simple and efficient. Its runtime overhead is one additional Forth branch compared to static :-definitions. And the compiler mechanisms take 30 lines of code.

>: <name> produces a :-definition prefixed by a branch to its code body. When <name> is compiled or executed, the xt for this branch instruction is compiled or executed and the actual code executed depends on the (re-writeable) branch destination.

>: leaves the branch destination's address and the address of the body of the current definition on the stack. Later, ; rewrites the branch destination thus modifying <name>'s semantics. EMBED allows to compile the previous and soon abandoned semantics of <name> into the new <name>.

Have fun!

Forth words

`>:          ( <name> -- ) "Method"`

Used in the form **`>: <name> .... ;`** to define words with late binding capability. >:-definitions can be redefined later on using **`>:`** iteratively. This late binding mechanism has a run-time overhead of only one branch instruction compared.to static :-definitions. >:-definitions do not modify the semantics of **`<name>`** immediately. This will be done by **`;`** and therefore, the previous semantics of **`<name>`** can be embedded in the new semantics of **`<name>`** using **`embed`**.

`embed       ( <name> -- ), I`

Used in the form **`embed <name>`** in method definitions while redefining **`<name>`** in a >:-definition. The previous semantics of **`<name>`** will be compiled into the new definition.

`BIND        ( xt <proxy name> -- ), I`

`BIND` assignes execution token **`xt`** to a proxy, which is embedded more or less deeply (by way of attributes) in an object.
`BIND` is a state smart word. When interpreted, it stores **`xt`** in the proxy field of an object. When compiled, it compiles the address of **`<proxy name>`** as a Literal followed by a store instruction that assignes **`xt`** to the proxy when executed later on.

`Class       ( <name> -- )`

Defining word. Based on Forth vocabularies, it holds a wordlist for the class's methods and keeps tabs of attribute sealed/unsealed, the size of an object, and a pointer to the parent class, which may have been inherited. By default, the parent class points to the ClassRoot wordlist, which is therefore inherited by every class.

When **`<name>`** is executed later on, it sets the class context. As with vocabularies, **`definitions`** can be used to add to its wordlist.

`ClassRoot ( -- )`

A wordlist of fundamental OOP words, which are accessible in every class.

`Method:     ( <name> -- )`

Used in the form **`Method: <name>`** in a class definition in order to create an "uninitialized method" **`<name>`**. Needed in order to create **`<name>`** for compilation into definitions before the actual method is defined later on using **`>: <name>`**.

Root words

`classes     ( -- )`

Lists all defined classes

`methods     ( -- )`

Lists the methods of the recent class.

`Self        ( -- ), I`

Sets the class context to the current class.

ClassRoot words

`New        ( <name> -- )`

Creates object **`<name>`** of the recent class.
`<name>` is a state smart word. When interpreted, it returns its object data field address and sets the class context.
When compiled, it compiles its data field address as a Literal.

Usually it is used in the form **`<classname> New <name>`**.

`::         ( <name> -- ) "Attribute"`

Create attribute **`<name>`** in the current class as an embedded object of the recent class.
Used in the form **`<classname> :: <name>`** in object definitions before sealing the class.
`<name>` is a state smart word.
When interpreted, it adds its data field offset to the object address on the stack.
When compiled, it compiles the offset as a Literal followed by + as operator.

`[]         ( <name> -- ) "Proxy"`
Create proxy **`<name>`** of the recent class as an immediate word. Used in the form **`<classname> [] <name>`** in object definitions before sealing the class. It reserves a field in class's objects for an execution token, which will be initialized with the " un-initialized reference" error. When **`<name>`** is executed later on, it executes code, which can be assigned using **`BIND`**. It resembles **`DEFER`** for objects.
`<name>` is a state smart word.
When interpreted, it executes the assigned code.
When compiled, it compiles the offset of **`<name>`**'s field in the object as a Literal followed by a word that fetches and executes the assigned code.

`inherit    ( -- )`
Used to fully embed the recent class in the current class. Used in the form <classname> inherit before specifying any other object field. The wordlist pointer, the size and the parent field of the current class will be set to **`<classname>`**'s values.

`units      ( u1 -- u2 )`
Used to compute the size u2 of allocating space needed for u1 object data fields of the recent class. Used e.g. in the form **`<classname> units allot`**.

`seal       ( -- )`
Used to finish the object definition of a class setting a bit in the class's attribute field. Used in the form **`Self seal`**. After sealing a class, the size of the data field of its objects has been established.

`addr       ( obj -- addr ), I`
Used in the form **`<objectname> addr`** as a universal method to switch back into the Forth context leaving **`obj`**'s data field address on the stack. <objectname> may consist of the name of an object followed by a sequence of attribute names.

`..          ( obj -- addr ), I`

Synonym for **`addr`**. For debugging.

`size        ( -- bytes )`

Universal method that returns the size of the data field of the recent class.