## Introduction

File OOP.FS holds the definitions for "simpleOOP", 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 - 415 lines of code realize all mechanisms, which can be expected from an OO package to the best of my knowledge, namely: Polymorphism, late binding, single inheritance and Proxies.

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 TEST.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 polymorphism (@ in String). File DEMO.FS is another test case file that concentrates on polymorphism 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 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 nor polymorphism. 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, and the solution I found is embarrassingly simple and efficient. Its runtime overhead is one additional Forth branch compared to static :-definitions. And Bernd Paysan argued that I couldn't possibly do without vTables for proper polymorphism. As it turns out, Bernd was both right and wrong: simpleOOP does not have classical vTables; instead, the wordlist associated with each class constitutes the vTable of that class. These "virtual" vTables are indexed by pattern-matching (searching for a name), the table entries are directly executable (which is unusual), and the dynamic binding element is a re-writeable branch. To preserve polymorphism on inheritance, all words of the superclass will be physically copied into the subclass's wordlist upon creation, much like you make a copy of the vTable.

Instead of creating new words for OOP handling, I tried to make existing words behave differently depending on whether they are used in a Forth or in a Class context. Therefore, there is e.g. no special defining word for methods. : just behaves differently, if we are compiling into a class wordlist. This helps to minimize the number of new words needed for simpleOOP and the resulting code looks very "forthish".

In a class compilation context, ": <name>" produces a :-definition, which starts with a branch to its code body. Therefore, the semantics of <name> depends on the (re-writeable) branch destination. Later on, when ": <name>" is redefined in order to e.g. instrument a method, no new header is created, but the branch destination of its first version is re-written. This is an efficient late binding mechanism.

Polymorphic methods have to be declared explicitly using "Polymorphic <name>". This allocates space for an XT in each instantiated object and its "default" behaviour can be specified using ": <name>" later on. In addition, an object specific behaviour can be assigned using bind. Polymorphic methods can only be created while a class is not yet sealed. It will be sealed automatically as soon as a) an object of that class is created, b) the class is embedded in another class as an attribute, or c) when a subclass is created.

## Forth words

**`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 (see below).

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

**`ClassRoot ( -- )`**

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

**`Polymorphic    ( -- )`**

Used in the form

**`Polymorphic <name>`**

to define polymorphic methods. At first, <**`name>`** will be associated with the "un-initialized method" error handler. Later on, its specific behaviour in its own class or in a sub-class can be defined by just defining a colon definition of the same name.  An object specific method can be assigned using **`BIND`**.

**`Oop ( -- )`**

a Forth vocabulary that holds all primitives, which are needed to implement simpleOOP.

## Root words

**`classes   ( -- )`**

Lists all defined classes

**`methods   ( -- )`**

Lists the methods of the most recently executed class and its inherited sub-classes.

## ClassRoot words

**`Object    ( <name> -- )`**

Used in the form **`<classname> Object <name>`**. Creates object **`<name>`** of class **`<classname>`**.
**`<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.

**`Attribute ( <name> -- )`**

Create attribute **`<name>`** in the current class as an embedded object of the most recent class as an immediate word. Used in the form **`<classname> Attribute <name>`** in object definitions before the class is sealed.
**`<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.

**`Proxy      ( <name> -- )`**

Create proxy **`<name>`** of the recent class as an immediate word. Used in the form **`<classname> Proxy <name>`** in object definitions before the class is sealed. 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 to an object 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.

**`bind       ( xt obj <proxyname> -- ), I`**

**`bind`** assignes execution token **`xt`** to a proxy or a polymorphic method, which has defined for object **`obj`**.
**`bind`** is a state smart word. When interpreted, it stores **`xt`** in the proxy / polymorphic method field of **`obj`**.
When compiled, it compiles the offset of **`<proxyname>`** in **`obj`** as a Literal followed by instruction **`do-proxy`** that assignes **`xt`** to the proxy or polymorphic method when executed later on.

**`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`**.

**`allot      ( u -- )`**

allocate u bytes space in objects of the current class.

**`size       ( -- bytes )`**

universal method that returns the size of the data field of the recent class's objects.

**`'          ( <name> -- xt )`**

returns the **`xt`** of **`<name>`** in the class context.

**`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.

**`definitions ( -- )`**

used in the form **`<classname> definitions`** in order to make **`<classname>`** the compilation class.

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

Synonym for **`addr`**. Used while debugging in order to escape the class context.

**`see        ( <name> -- )`**

de-compiles <name>.

**`order      ( -- )`**

displays the actual class context.

**`methods   ( -- )`**

used in the form **`<classname> methods`** to list the methods of class **`<classname>`** and its inherited sub-classes.

**`words     ( -- )`**

used in the form **`<classname> words`** to list the methods of class **`<classname>`**.