Oforth Programming Language

Manual

Franck Bensusan / Hugo Mlodecki

Table of contents

1 Introduction....6
1.1 What is Oforth ?....6
1.2 Words naming conventions....7
1.3 Installation....7
1.4 Invoking interpreter....8
1.5 Running programs....10

2 Interpreter and data stack....11
2.1 Interpreter....11
2.2 Data stack....11
2.3 RPN notation....12
2.4 Data stack and objects....13
2.5 Word stack effects....14
2.6 Manipulating the stack....15

3 Arithmetic....17

4 Functions and instructions....19
4.1 Declaring a function....19
4.2 Flow control....20
4.3 Comparisons....21
4.4 General loops....22
4.5 Return stack and locals....23
4.6 Integer loops....25
4.7 Recursion....26
4.8 Returning from a function....26
4.9 A (little) transgression to RPN notation....26
4.10 Factoring....27

5 Basic types....29
5.1 Object29
5.2 Null 29
5.3 Number (parent of Integer, Float)....30
5.4 Integer....30
5.5 Boolean....31

5.6 Character....32
5.7 Float 33
5.8 Block and anonymous functions....34
5.9 Symbol....35

6 Object Oriented Programming....36
6.1 Introduction....36
6.2 Classes and attributes....37
6.3 Methods and implementations....39
6.4 Polymorphism....41
6.5 When no hierarchy is better than bad hierarchy....43
6.6 Properties : when no hierarchy is better than bad hierarchy... again....44
6.7 Polymorphism revisited....45
6.8 Function or Method ?....46

7 Dictionary and OO meta-model....48
7.1 Words48
7.2 Constants....49
7.3 Task variables....49
7.4 Aliases....50
7.5 Deferred words....50
7.6 Other words....52

8 Compilation....53
8.1 The current definition....53
8.2 Interpreter state....53
8.3 Native code optimizations....55
8.4 User defined immediate functions....56
8.5 Interpreter revisited....57
8.6 The Control Stack....57
8.7 Directives....58

9 Higher order functions and collections....59
9.1 #forEachNext method and #forEach: function....59
9.2 Arrays....60
9.3 Higher Order Functions....61
9.4 Indexable collections....63
9.5 Mapping collections and ArrayBuffers....63

9.6 Mapping collections..... 64

10 Collection classes..... 67
10.1 Pair 68
10.2 Hash..... 68
10.3 Interval..... 68
10.4 Stack..... 69
10.5 Json 69
10.6 Buffer..... 70
10.7 MemBuffer..... 71
10.8 String..... 71
10.9 StringBuffer..... 73

11 I/O and formatting..... 74
11.1 Formatting objects..... 74
11.2 Files 76
11.3 Basic input/output..... 79
11.4 Console..... 79

12 Environment..... 81
12.1 Environment constants..... 81
12.2 Functions..... 81

13 Concurrent programming..... 83
13.1 Immutability and task isolation..... 83
13.2 Threads and workers..... 84
13.3 Tasks..... 84
13.4 Channels..... 85
13.5 Resources..... 87
13.6 Tasks reporting..... 88
13.7 Summary..... 89

14 Memory..... 90
14.1 Memory areas..... 90
14.2 Allocating/freeing memory..... 90

15 Exceptions..... 92
15.1 Catching exceptions..... 92
15.2 Exception class..... 93

15.3 Predefined exceptions..... 93

16 Packages..... 95
16.1 Loading a package..... 95
16.2 Search for packages..... 96
16.3 Search order for words..... 97
16.4 Creating a package..... 97

17 Structures and FFI..... 99
17.1 Structures..... 99
17.2 Dynamic Libraries..... 100
17.3 Dynamic functions..... 101

18 Ans Forth / Oforth cross reference..... 102

1 Introduction

1.1 What is Oforth ?

Oforth is a Forth dialect. Objective is to implement an Object Oriented Programming model as a built-in feature (and not as an extension).

Even if not compatible with classical Forth (some Oforth features are too far from classical Forth), most Forth principles are preserved :

- Oforth is a stack based language : a data stack holds parameters for functions and methods and holds values returned.
- A return stack holds the return address of calls, local parameters and variables values.
- A dictionary holds Oforth words. In Oforth, there are many words types. Among new word types, you find classes, methods, properties ... that implement Oforth OO meta-model.
- The inner interpreter is updated to, in addition to word execution, handle polymorphism for methods, with late binding.
- Oforth is an interpretive language : the text interpreter execute input and definitions are compiled using immediate words.

The first goal of Oforth language is to have an intuitive OOP implementation ie being able to call methods exactly the same way you would call classic words. There is no special syntax, no current object : when a method is called, the inner interpreter checks the object on top of the stack and calls the method implementation this object respond to. That's all : calling a method is no more complicated than calling a function and many Forth words are actually implemented as Oforth methods (like +, -, ...).

The second goal is to have a very light and open OO model. OO can sometimes be cumbersome and comes with many constraints. In Oforth, you can program without creating classes or without knowing that some words are methods and not functions.

The third goal is to implement an OOP model as "pure-OO" as possible, which means that :

- Everything in Oforth is an object, even OO meta-model objects : functions, methods, classes, ...
- Oforth implements a dynamic dispatch: the process of selecting which implementation to run according to the top of the stack is (unless optimized during compilation) done at runtime.
- Oforth implements duck typing, which means that any class, regardless of class hierarchy, can implement any method.
- Memory is handled by a garbage collector. Oforth implements an incremental mark and sweep collector.

And, last goal, keep high performances. This is done by generating and optimizing native code on the flow while keeping a one-pass compilation.

1.2 Words naming conventions

Oforth built-in words follow some naming conventions. The main conventions used are :

Classes and properties name begins with an uppercase :

```
Integer Array Comparable
```

Functions/methods names generally begin with a lowercase and with an uppercase for each word :

```
dup isKindOf name isA isDigit toUpper
```

Constants are often all uppercase. If related to a particular class, the name begins with this class and a dot :

```
File.READ System.VERSION CELLSIZE
```

Functions/methods that parse the input and/or need something after (which is not the natural RPN notation) have a name ending with ':' :

```
new: loop: : method: try: const: tvar: ifTrue: else:
```

Functions/methods can have a prefix to describe its purpose :

```
as : conversion      : asInteger asFloat, asString, ...
is : test the object and returns a boolean : isEven isOdd
if : test the top of stack : ifTrue: if=:
```

Those rules are not mandatory, but this gives important information when reading code.

1.3 Installation

Oforth 32bits runs on 32bits or 64bits operating systems.

Oforth is coming as an archive file that contains everything necessary to run Oforth :

- Oforth binary : it is the only executable into the archive (no library, ...)
- oforth.of : the file loaded at startup.
- lang dictionary : all features loaded at startup.
- packs dictionary : optional packages.

Installation requires two main steps :

- Extract the zip file into a directory of your choice.
- Define an environment variable : OFORTH_PATH

Oforth requires the environment **OFORTH_PATH** variable to be created and set. You can find instructions on the web if you don't know how to create an environment variable on your specific system.

OFORTH_PATH value is a list of directories. Oforth will try to find sources and packages into those directories. It should at least include the directory where you have extracted Oforth archive.

On Windows platforms, this value is a list of directories with ' ;' character as separator and '\' character for directory names. For instance :

```
OFORTH_PATH=\Home\Oforth;\Home\Oforth\test
```

On Linux and Mac OS platforms, this value is a list of directories with ':' character as separator and '/' character for directory names. For instance :

```
OFORTH_PATH=/users/oforth/oforth
```

You can also have this variable set each time you launch a command prompt. Instructions can also be found on the web according to your system.

Finally, if you would like to run Oforth from everywhere, you can also add your installation directory to your PATH variable.

Before running Oforth, check that OFORTH_PATH variable correctly set. On Windows, open a command prompt and run :

```
set
```

On Linux or Mac OS, open a command prompt and run :

```
env
```

The OFORTH_PATH variable must appears with the correct value to be able to run Oforth.

Currently, Oforth is a 32bits application so, if libraries are used, the 32bits versions must be installed if not present.

1.4 Invoking the interpreter

Oforth is invoked from a command line. Interpreter is launched using "--i" command line option:

```
oforth --i
```

This command will start Oforth in interpreter mode: you can directly type commands and see results. You can leave the interpreter using **bye** command.

Various command line options can be set when launching Oforth:

```
oforth [ options ] [file]

Launch options :
   --i    : Interpret mode
   --P"s" : Performs string s
   [file] : Performs code into file.

Behavior options :
   --a   : Assertions are checked (default is not checked)
   --t   : Tests are checked (default is not checked)
   --C   : Generates various checks (stacks underflow, overflow, ...)
   --U   : Run unsecure oforth ie allow raw access memory
   --Wn  : Max number of workers (default is number of cores)
   --Sn  : Max size (in objects) for main data stack (default is 1000 objects)
   --Mn  : Max memory to use by oforth process (Ko)

Garbage Collector options :
   --XTn : Set number of milliseconds between 2 GC (default is 120 ms)
   --XMn : Set min allocated memory (Ko) for GC to run (default is 1024 Ko)
   --XGn : Set GC ticks by step during mark & sweep phase (defaut is 6000)
   --XAn : Set app ticks by step during GC (defaut is 300)
   --XVn : Set GC verbose level (0 to 3, default is 0)
```

Examples :

```
oforth --i
Launch Oforth interpreter : you can execute commands until bye is typed.

oforth myfile.of
Launch Oforth, load myfile.of file and exit.

oforth --P"test" myfile.of
Launch Oforth, load myfile.of file, run "test" word and exit.

oforth --i --P"myfile.of load"
Launch Oforth interpreter, load myfile.of and prompt for commands.

oforth --i --P"import: date"
```

```
Launch Oforth intepreter, import date package and prompt for commands.
```

1.5 Running programs

As Oforth comes with an interpreter, you can test interactively your code. But, at one time, you will want to keep your work into a file and load it at startup. You don't have to compile anything, you can just give your file to Oforth program to load and execute it.

By convention, Oforth sources are stored into .of files, but this is not mandatory.

If you want to load test.of file and then run #test function :

```
oforth --P"test" test.of
```

But, in order to test your code, it is often more convenient to load your file(s), and execute tests into the interpreter. To do this, you can load your file into the interpreter :

```
oforth --i
"test.of" load
```

If you update your source files, you must leave the interpreter and reload the file. To avoid loading your file(s) each time you launch the interpreter, you can load it using the command line :

```
oforth --i --P"\"test.of\" load"
```

Then, you will launch the interpreter with your file(s) already loaded, ready to test your code.

And, a time will come when you will have packages (see Packages chapter). You can do the same thing and load you package before launching the interpreter :

```
oforth --i --P"import: mypackage"
```

You can also use Oforth as a pipe or redirect input or output

```
oforth < source.of >result.txt
```

After initialization, the interpreter will load "source.of" file and redirect results to "result.txt" file. After the source file is loaded, the interpreter will quits. Unlike interpreter mode, it will also quit immediately if a non catched exception occurs.

2 Interpreter and data stack

This chapter explains how the interpreter works, the concept of data stack and how to manipulate it.

2.1 Interpreter

When you launch Oforth with --i option, the interpreter is launched : you can play with the stack, create words, create methods, load files, ... Oforth is an REPL system (Read Eval Print Loop)

Interpreter is very simple: after you type something and press ENTER key, it reads the first name (by collecting all characters until a space is encountered) and executes it. After this name is executed, the interpreter reads the next name and execute it. And so on, until there is no more name to execute (or the word executed is **bye**). There is no instructions, just words separated by spaces.

A name is a sequence of characters terminated by a space (or end of line). An important difference with most Forth interpreters is that Oforth interpreter will also try to identify some names even if there is no space to delimit them. Those names are:

```
# @ { } , : ; ( ) ! $ [ ] | ‘ " // \ -> #[ => 0x 0b #! :=
```

When the interpreter has found a name (either because of a space or because it is one of the words listed above), it tries to find if it is the name of an existing word (a built-in word or a user-defined word). For this, it searches this name into the **dictionary**, a place where all words are stored. If a word with this name is found, the interpreter executes it and handles the next name. If the name is not found, the interpreter tries to detect if the name is an integer or a float. If so, it pushes this number on the stack. And, if the name is not a number, then an error is printed and the rest of the line is ignored.

Interpreter is case-sensitive: for instance, if you try **Bye** or **BYE**, interpreter will not recognize the word **bye**.

2.2 Data stack

Oforth is a stack based language. Parameters needed by a word are sent using a stack, prior to calling the word. Those words push also (if any) their return values on this stack,. This stack is called the **data stack** (or ... the stack).

The data stack is a LIFO stack : the last pushed item will be the first popped item. All words have access to this stack and each time you create an object or type a number, it is pushed on the data stack.

There is only one data stack (in fact, one by task in a multi-tasking context), whatever the kind of object : integers, floats, strings, all objects are handled by the same data stack. This is a difference with Forth Standard where floats have a dedicated stack.

Each word performed will retrieve its parameters (if any) on this stack, does its job and returns its values on the stack. So each word has an effect on the stack, which is the way it interacts with the stack. For instance:

```
sqrt          \ f -- f'
```

means that the word sqrt is expecting a float on the stack, consume it and, when it has done its job, pushes a float on the stack (\ word is for declaring a commentary till the end of the line).

So you can try :

```
12.3 sqrt
```

The interpreter will push float 12.3 on the stack and then call word sqrt.

Using the data stack is a very different paradigm than how other languages work : parameters are not named and words take and give parameters to others words by using the data stack.

```
1.2 ln sqrt exp
```

Here, each word uses the result of the previous one as its parameter and the result is on the stack. When reading this sequence, nothing tells you what are the parameters each word uses. The stack is the quiet support for passing parameters between each word.

The data stack is one of the most important concepts in Oforth. If you don't use correctly the data stack, your code will be longer and more complex than necessary. Mastering the usage of the data stack is an important step.

2.3 RPN notation

Because the interpreter executes each word before reading the next one, the interpreter uses the **RPN notation**.

In this notation, there is no need for parenthesis: all parameters for a word are pushed on the stack prior to execute the word. For instance, in order to calculate 1 + 2, you write:

```
1 2 +
```

+ consumes two parameters on stack, calculates the sum, and pushes the result on the stack :

```
+             \ n1 n2 -- n3
```

To see the stack, you can use **.s** word

```
1 2 + .s
[1] (Integer) 3 ok
```

You can also consume and print the top of the stack using **.** word :

```
1 2 3 + . .s
```

RPN notation works very well with OOP : a method is sent to an object, so a way to do it is to push this objects on the stack and to call the method. Methods just apply to the object on top of the stack. RPN notation performs well with OOP and is quite natural to use.

```
1.2 ln .s
```

2.4 Data stack and objects

Oforth is a dynamic typed language : each data is typed and the data stack holds typed objects.

Each object, whatever its size, uses one and only one position on the stack. An integer, a float, a string, a date, a collection of 1000 objects, ... correspond to one position on the stack.

Objects themselves are created in the heap (or in the dictionary) and the data stack holds references to those objects. All words that manipulate the stack items, manipulate references to objects (an important exception to this rule is small integers, see Integer chapter).

Each object has a type : its class. When you type **.s**, the object's class is what appears between (), before the object value. For instance:

```
1 2.3 Integer "abcd" .s
```

You can toggle interpreter to automatically show the stack after each command you enter using .show command . If so, a .s will run after each command executed. .show again will go back to the previous behavior (not show the stack).

```
.show          \ --          Toggle intepreter to show the stack after each command
bye            \ --          Leaves the interpreter.
```

Advanced: objects and data stack structure

On Oforth 32bits, a data stack cell is 32bits long and can hold a value between 0 and 4 Gb. This range is split in two: 2 Gb for objects references created on the heap and 2Gb for integers.

This means that an Oforth 32bit process can't allocate more than 2 Gb of memory. This is not a real constraint as, on some 32bits OS (Windows for instance), a process can't allocate more than 2Gb, no matter how RAM is available (the value is 3Gb on 32bit Linux).

This also means that small integers are into [-1Gb, 1Gb] range. This is not a big constraint too, as integers overflow are detected and integers switch automatically and transparently to arbitrary precision integers if needed.

Of course, there is not such constraints on Oforth 64bits.

2.5 Word stack effects

When a word is executed, it has an effect on objects on the stack : it can consume parameters and can return value(s).

A word's stack effect describes how this word interacts with the stack : it is a commentary to document this interaction :

```
\ stack before -- stack after
```

For instance :

```
foo             \ n1 n2 n3 -- n4 f
```

means that word foo, when performed, consumes 3 objects (here 3 integers) on the stack and returns two objects (here an integer and a float) on the stack. This also means that the action of foo does not affect items under n1.

With this notation, n3 is the top of the stack before foo is performed and f is the top of the stack after foo is performed : the top of the stack is always the rightmost element.

It is a good practice (and required for readability) to describe the stack effect of each word.

For instance, the stack effect of - arithmetic operation on integers is:

```
-               \  n1 n2 -- n3   with n3 = n1-n2
```

Conventions used to describe stack effects are:

- x, y, z An object, whatever its type is.
- n An signed integer
- u An unsigned integer
- f A float

- b A Boolean **(true** or **false)**
- c A character (ie the unicode value of a character).
- s A string
- [x] A collection of objects.
- r A runnable (ie something you can perform, like functions, methods, blocks, ...).
- rcond A runnable that returns true or false when applied on an object.
- cl A class
- ex An exception
- "name" A name is read from the input buffer.

2.6 Manipulating the stack

There are various words to manipulate items on the stack. Those manipulations are used to order items on stack before calling functions and methods.

These words manipulate only items references. If you duplicate an object, you duplicate only its reference value, you don't create a new object.

The most important words that manipulate the stack are :

```
dup             \ x -- x x          Duplicates the top of the stack
drop            \ x --              Removes the top of the stack
swap            \ x y -- y x        Swap the two items on top of stack
over            \ x y -- x y x      Copy the second item on top of stack
rot             \ x y z -- y z x    Rotate 3 items on the stack
```

Other less used words are :

```
nip             \ x y -- y          Remove the second item
tuck            \ x y -- y x y      Copy the tos under the second (same as swap over)
-rot            \ x y z -- z x y    Rotate 3 items
2dup            \ x y -- x y x y    Duplicate 2 items
2drop           \ x y --            Remove 2 items
pick            \ x*i n -- x*i xn   Copy the nth item on top ( 1based )
.depth          \ x*i -- x*i n      Return stack size
under+          \ x y a -- x+a y    Adds a to the second item.
under++         \ x y -- x+1 y      Increments the second item.
under--         \ x y -- x-1 y      Decrements the second item.
```

Examples :

```
1 2 3 swap          \ 1 3 2
1.1 2.3 over        \ 1.1 2.3 1.1
```

```
"aa" "bb" tuck       \ "bb" "aa" "bb"
12 14 2dup           \ 12 14 12 14
1.3 2.3 .depth       \ 1.3 2.3 2
```

3 Arithmetic

Oforth implements arithmetic operations as methods : the same method name is used for all types. Operator + is defined for integers, floats, strings, arrays, ...

There is also a priority between four types :

```
Integer < Float < String < Array
```

If an operator is to be executed on two items with different types, item of lower priority will be automatically converted into an object of the other type.

Arithmetic words are:

```
+           \ x y -- x+y       Implemented for numbers, strings, arrays
-           \ x y -- x-y       Implemented for numbers, strings, arrays
*           \ x y -- x*y       Implemented for numbers
/           \ x y -- x/y       Implemented for numbers
neg         \ x -- -x          Implemented for numbers
sgn         \ x -- n           Implemented for numbers, returns -1, 0 or 1
abs         \ x -- y           Implemented for numbers
sq          \ x -- x*x         Implemented for numbers
pow         \ x n -- x^n       Implemented for numbers
^           \ x n -- x^n       Same as pow.
powf        \ x f -- x^f       Implemented for numbers
ln          \ x -- f           Implemented for numbers
log         \ x -- f           Implemented for numbers
exp         \ x -- f           Implemented for numbers
sqrt        \ x -- f           Returns a float
nsqrt       \ n -- m           Returns integer m / m*m <= n < (m+1)*(m+1)
mod         \ n m -- r         Returns the remainder of n by m
/mod        \ n m -- r q       Returns the remainder and quotient of n by m
```

Some words are shortcuts for usual operations:

```
1+          \ x -- x+1         Add 1 to x
1-          \ x -- x-1         Substract 1 to x
2*          \ x -- x*2         Multiply x by 2.
```

```
2/            \ x -- x/2         Divide x by 2.
```

All arithmetic operators use the RPN notation so no parentheses are necessary :

```
1 2 + 3 * .s         \ Calculates   (1+2)*3
1.2 ln 4 + exp .s    \ Calculates   exp((ln(1.2)+4))
```

Advanced topic : type conversion

In order to convert an object, a method uses a convertor. A convertor is a method which name is "as" + the type for which we need a conversion.

If "1 2.3 +" is typed, the sequence will be "1 asFloat 2.3 +"

If "2.3 1 +" is typed, the sequence will be "2.3 1 asFloat +"

If "1 [2, 3, 4] +" is typed, the sequence will be "1 asArray [2, 3, 4] + "

That is why you can encounter exceptions like "null does not understand #asInteger". It means that you are trying to run an operator between an integer and null. The method tries to convert null into an integer, but asInteger is not defined for null so an exception is raised (btw, asInteger is not defined for null on purpose, in order to detect this kind of error).

4 Functions and instructions

As for many languages, functions are a named piece of code. Calling a function is done by typing its name. In Oforth, functions are objects representing the classical Forth words.

4.1 Declaring a function

A function is declared using a colon (the word :). The word ; closes the function definition. Everything between : and ; is the function body (the current definition) : it is the instructions that will be performed when the function is called.

```
: helloWorld         \ --
    "Hello, world!" . ;
```

This creates a function named helloWorld into the dictionary. The stack effect shows that it takes no parameter from the stack and returns no value, so whatever was on the stack before calling this function is not modified. helloWorld pushes a string on the stack, then calls function . which stack effect is (x --) : it takes something on the stack and sends it to the standard output.

Calling helloWorld function is simple : you just type its name. It can be called directly into the interpreter (to test it, for instance), or called from another function :

```
helloWorld


: test        \ --
    helloWorld ;
```

A function is not only a named piece of code; it is also an object of type Function stored into the dictionary. This object can be manipulated like any other object : it can be pushed on the stack, used as parameter for other words, ... To push this object on the stack, # is used before the function name. This word reads the next string and retrieves the word which name is this string.

```
#             \ "name" --


#helloWorld .s
```

No space is needed after # (but you can type one if you want) , as # is a function listed in the previous chapter as detected by the interpreter.

After pushing a function on the stack, it can be used as any other object. In particular, #perform is a method that executes the top of the stack :

```
#helloworld perform
#helloworld #perform #perform perform
```

Advanced topic: execution tokens, named tokens and function objects.

In Oforth, there is no xt (execution token) as in Forth : a function is an object that allows to retrieve all information about it. This object is close to a named token. The ‘ Forth word that allows to retrieve an execution token from a word is replaced by # word, which returns an object on the stack. ‘ word is dedicated to characters.

As there is no execution token, their is no EXECUTE word. It is replaced by #perform, which performs the code associated to the function”s object.

Into the rest of this manual, a function (or method) will often be mentioned using # to show that we are talking about a function.

4.2 Flow control

A function body can use conditional structures to control the instructions flow. These structures consume a boolean on the stack and execute instructions according to its value.

Booleans don't have a dedicated type, they are implemented as integers. They are the constants **true** (value 1) or **false** (value 0). In any test, everything different from **false** is considered **true**.

Advanced topic : true value is not -1

In Oforth, true value is 1 and not -1.

As integers have arbitrary precision, a -1 value is not 0XFFFFFFFF or 0xFFFFFFFFFFFFFFFF

Various structures are available to condition instructions. They all work the same way : they consume a boolean on the stack and condition instructions according to its value.

```
: myabs        \ n1 -- n2
    dup 0 <= ifTrue: [ neg ] ;
```

#<= consumes two objects and returns a boolean on the stack. This boolean is consumed by ifTrue: . If this boolean is not false, all instructions between [and] are performed. Otherwise, the program jumps directly after].

Other words to test conditions are:

```
ifTrue: [ instr ]     \ b --       Instructions are performed only if b is not false
ifFalse: [ instr ]    \ b --       Instructions are performed only if b is false
ifZero: [ instr ]     \ x --       Instructions are performed only if x is 0
ifNull: [ instr ]     \ x --       Instructions are performed only if x is null
ifNotNull: [ instr ] \ x --       Instructions are performed only if x is not null
if=: [ instr ]        \ x y --     Instructions are performed only if x = y
```

All these blocks can be followed by an **else** block. If so, this block is performed only if the instructions are not performed.

```
: mysign              \ x -- n
    dup 0 < ifTrue: [ drop -1 ] else: [ 0 > ] ;
```

The classical **if/else/then** conditionals used in Forth are also present :

```
: myabs               \ x -- y
    dup 0 <= if neg then ;


: mysign              \ x -- n
    dup 0 < if drop -1 else 0 > then ;
```

4.3 Comparisons

Two different objects can have the same value (two strings, two floats, ...). It is needed to differentiate if we want to test objects values or objects references.

The word **#=** checks if two objects are the same object (ie have the same reference) :

```
=                     \ x y -- b : Returns true if x and y are the same object


12 dup =              \ true
12 12 =               \ true
1.2 dup =             \ true
'a' 'a' =             \ true
1.2 1.2 =             \ false
"aaa" dup =           \ true
"aaa" "aaa" =         \ false
#dup #dup =           \ true
[ 1, 2 ] [ 1, 2 ] =  \false
```

In other words, #= tests equality of the values of the two cells on top of the stack, whatever the objects are and whatever their type.

On the other hand, the method #== tests if two objects have the same value. By default, this method calls #= , but it can (and sometimes must) be redefined into classes to test objects values rather than object's references.

```
==                      \ x y -- b : Returns true if x and y have the same value


12 12 ==                \ true
"aaa" "aaa" ==          \ true
1.2 1.2 ==              \ true
[ 1, 2 ] [ 1, 2 ] ==    \ true
```

#== is an operator : if the two objects don't have the same type, the one will the lower priority will be converted.

```
12 12.0 ==              \ true
```

There is no word to test if two references are different but there is a word to test if two objects don't have the same value : #<>

```
<>                      \ x y -- b : Returns true if x y don't have the same value
```

4.4 General loops

Three structures allow to loop according to boolean values.

The first one is an infinite loop **begin/again** : all instructions between those words will be performed endlessly, unless an instruction orders to leave the loop (**return**, **break**, ...).

```
: test1       \ n --
    begin
        dup 100 = ifTrue: [ break ]
        1+ dup .
    again
    "Done" .
;
```

The second (and the most used) loop is the while loop. While a boolean value is true, instructions are performed :

```
while ( ... b ) [ instructions ]
```

Instructions between (and) must leave a boolean on the stack. This boolean is consumed. If it false, the program leaves the loop. Otherwise, instructions are performed and the program jumps back just after the while :

```
: mygcd       \ n1 n2 -- n3
    while ( dup ) [ tuck mod ] drop ;


120 32 mygcd .
```

The last loop is the doWhile loop. Unlike while, instructions are always performed at least once :

```
doWhile: [ intructions b ]
```

Instructions are performed and must leave a boolean on the stack. This boolean is consumed and while it is not false, instructions are performed again.

Some words allow modifying the flow into a loop : if **break** is encountered, the program leave immediately the current loop and if **continue** is encountered, the program restart immediately the current loop :

```
: test2       \ --
    10 while ( dup ) [
        1-
        dup 6 = ifTrue: [ continue ]
        dup 3 = ifTrue: [ break ]
        dup .
        ] drop
;
```

4.5 Return stack and locals

When a function calls another function, the return address must be saved. This is done on a second stack, the **return stack**.

Each call to a function creates a frame on the return stack. And, when the function returns, its frame is removed from the return stack. By default, this frame holds only the return address of the function, but it also allows to save information local to a function call. For instance, to avoid many data stack manipulation, it is possible to store a value on the return stack. As the entire frame is removed from the return stack, this value will be lost when the function returns.

Storing local information on the return stack is done by declaring a **local** into a function. Two kind of locals can be declared : parameters and local variables.

If a function declares a parameter, when this function is called, an object is removed from the data stack and stored on the return stack as the parameter value. Using the parameter name into the

body will push this value on the data stack. To store another value (the top of the stack) into this parameter, #-> word is used.

Parameters are declared using ( ) after the function name. All names until) are declared parameters. #-- is the beginning of a commentary and used to describe stack effect while declaring parameters. Everything between -- and) is ignored. So it is possible to mix the parameters declaration and the stack effect description of a function, avoiding to create a separated commentary to describe the stack effect.

CAUTION (for Forthers)

In Oforth, parentheses () are not commentaries, they are used to actually declare parameters.

{ } are used for another purpose, to declare Json objects.

For instance :

```
: sumDigits( n -- m )
    0 while( n ) [ n 10 /mod ->n + ] ;

123 sumDigits .
```

This function declares one parameter (n). Text between -- and) is the commentary that describe the stack effect of this word. When this function runs, it removes an object from the stack and stores it on the return stack into the frame created when the function is called. Each time n is used into the function's body, the current value stored on the return stack is pushed on the stack. And -#> is used to change the value of n (by removing an object from the stack and storing it on the return stack).

If more than one parameter is declared for a function, the last one will have the value of the top of the stack, then the previous one, ... Parameters are declared in the same order than stack effects. For instance, #under+ consumes and adds the top of the stack to the second item of the stack :

```
: under+( x a )      \ y x a -- y+a x
   a + x ;
```

A function can also declare local variables. While parameters values are initialized with objects removed from the stack, local variables values are initialized to null value on the return stack. This is the only difference : using a local variable name will push its value on the stack and #-> will set its value with an object on the stack. Local variables are declared after parameters and before the function body using #| word.

```
: mybench            \ r -- ...
| tick |
   System.tick ->tick
```

```
perform
System.tick tick - . ;
```

Locals usage are subject to discussions : if you use too much locals, you will miss the goal of factoring your words and you will miss the objective to master the data stack usage. But, sometimes, locals can avoid too many stack juggling (many swap, dup, ...). What you have to know about local is that they are fully optimized and there is no reason to not use them just for performance purposes. Just be careful to miss to factor your code because of use of locals. Oforth is not C : function body are almost never more than 3 lines long. Locals could help you to maintain longer functions, but you can miss good factorization. This is particularly true for programmers coming from C-like languages : using too much locals will block your learning of good use of the data stack and factorization art.

Advanced topic : return stack implementation

In Oforth, the return stack is not directly accessible like in a classical Forth (there is no words such as >R, <R, I, J, ...). There are two reasons : first, this allows to optimize access to locals and, second, the receiver of methods is also stored on the stack and handled automatically when a method returns.

No memory is allocated for the return stack : it is implemented directly on the thread stack. Code generates calls function using CALL assembler opcode and returns from functions using RET assembler opcode. On the other hand, the data stack is allocated on the heap.

4.6 Integer loops

Integer loops use an index and will run a loop for each value of this index between a range. All integer loops need a local variable to be declared into the function.

```
: fact        \ n -- n!
   | i | 1 swap loop: i [ i * ] ;


500 fact .
```

#loop: consumes an integer on the stack and runs instructions between [ ] for each value between 1 and this integer. Into the block, the local value is this current value.

#for: is similar to **#loop:** but consumes 2 integers on the stack : instructions will run for each value between those 2 integers (included).

```
: test        \ --
   | i | 10 20 for: i [ i . ] ;
```

Loop can also go backward using **#-loop:** word :

```
: test        \ --
    | i | 10 -loop: i [ i . ] ;
```

4.7 Recursion

Into a function's body, it is possible to directly call this function to implement a recursion :

```
: fact        \ u -- u!
   dup ifZero: [ drop 1 ] else: [ dup 1- fact * ] ;


10 fact .
```

Another example with Fibonacci sequence (and declaring one parameter) :

```
: fib ( n -- m )
   n 1 <= ifTrue: [ 1 ] else: [ n 1- fib  n 2 - fib  + ] ;


10 fib .
```

4.8 Returning from a function

It is possible to return immediately from a function before the end using **#return** word. The return value(s) is what is on the stack when #return is performed.

```
: test        \ n1 -- n2
    dup 3 = ifTrue: [ return ]
    4 + ;
```

The return stack is not "open" and is accessible only using locals, even for integer loops. So there is no restriction to return from a function. Removing a function's frame from the return stack when the function returns is handled automatically.

4.9 A (little) transgression to RPN notation

The interpreter always uses RPN notation. This notation is in the heart of the system (see Compilation chapter).

But, sometimes, when a function has many parameters or if parameter(s) are calculated, it can be

interesting, for readability purposes, to have a notation to separate those parameters. There is a "sugar" notation for this. () allows to push parameters after the function. Of course, this sugar is never mandatory.

```
: fib ( n -- m )
    n 1 <= ifTrue: [ 1 ] else: [ fib( n 1- ) fib( n 2 - ) + ] ;
```

Here, #fib has been rewritten to use this notation for the two inner calls : #fib parameters (and how they are calculated) are now clearly identified.

This notation has no runtime cost; when you write this version of #fib, interpreter translates it into the previous version.

This notation has nothing to do with how #fib function was declared (with or without declaring a parameter). As it is only sugar, it can be used for calling a function in either case.

This notation is also possible if a function takes more than one parameter :

```
: diag        \ a b -- x
    sq swap sq + sqrt ;
: test1       \ -- f
    diag( 2, diag ( 3, 4) ) ;
: test2       \ -- f
    2  3 4 diag  diag ;
```

#test1 and #test2 not only return the same value but the code generated is also the same.

Last point : it is possible to use this notation even at the interpreter level, ie you can type :

```
10 fib .
```

or

```
fib( 10 ) .
```

4.10 Factoring

Oforth programming is building your program by writing functions that have a contract with the data stack : parameters removed and returns values.

Functions should be very small (no more than 3 ou 4 lines) because, with bigger functions, it becomes hard to follow what happens on the stack. Don't be afraid to write small words, even words that call no more than 3 or 4 words.

Oforth programming is all about finding the good, small, reusable, named factors, to define them as functions and call them into other functions.

You can test those factors very easily by calling them directly at the interpreter level. And you should test your factors as soon as you write them.

As immutability is enforced, most of the time your function will have a very important characteristic : every time you send them the same parameters, they will return the same value(s), without side effects. This means that, when you test your functions once, they will work forever. Immutability also makes your functions thread safe.

The choice of function's names is very important. If you can't find a good name for a piece of code, it is probably not a good factor. Finding good names if much more important than in other languages : as the space is the word separator, good names make your code much more readable.

Factoring is a key concept to write good code. And the best possible factor is a small, simple, namable, with no side effect function.

Last point : factoring is not something frozen that ends when you have finished to write your code. If you write some code, re-read your previously written functions. The code you just wrote today can make you think of new factors for the code you wrote a week ago.

5 Basic types

This section describes the basic types declared at startup.

5.1 Object

Object is the top of the class hierarchy (superclass of Object is null).

Words implemented at this level are available for all objects, whatever their type. Many words are higher order functions and will be described into the dedicated chapter. Other words are :

```
yourself            \ x -- x : Returns the receiver
class               \ x -- aClass : Returns an object's class
isNull              \ x -- b : Returns true if x is null.
notNull             \ x -- b : Retruns true if x is not null
isA                 \ cl x -- b : Returns true if x class is cl
isKindOf            \ cl x -- b : Returns true if x class is cl or a child of cl
respondTo           \ m x -- b : Returns true if x responds to method m

==                  \ x y -- b : Checks if two objects have the same value (virtual)
<>                  \ x y -- b : Checks if two objects don't have the same value.

<<                  \ aStream x -- aStream : Send x to aStream
<<n                 \ aStream n x -- aStream : Send x to aStream n times
```

Some classes or properties add new words at the Object level (isComparable, isString, ...).

5.2 Null

Null is the class of the **null** object.

null object means nothing. It is used :

- To initialize a newly allocated object attributes.
- To initialize local variables values.

- As the return value of some function when we want to express that the function returns nothing.

Two functions exist to check if an object is null or not :

```
isNull        \ x -- b : returns true if x is null.
notNull       \ x -- b : returns trus if x is not null.
```

Null object is unique in the system and this object is not allocated on the heap.

5.3 Number (parent of Integer, Float)

The class Number is a representation of mathematical numbers. It is the superclass of Integer and Float. Other numbers can also inherit from Number classes (complex, ...).

Basic operations (+, -, *, /) are implemented in each Number' subclass.

The following words are implemented at the Number level :

```
abs                \ x -- y : Returns absolute value of x
neg                \ x -- y : Returns the opposite of x
sq                 \ x -- y : Returns x*x
sgn                \ x -- n : Returns number sign ( -1, 0 or 1 )
```

Others methods are added to Number class into other classes (Float for instance). This allows to apply the following words to all numbers :

```
sqrt inv root ln log exp
```

5.4 Integer

Integers have arbitrary precision.

On 32bits systems, Integers between [-1073741823, 1073741823] = [$-2^{30}+1$, $2^{30}-1$] don't allocate memory and are stored directly on the stack. Other integers are allocated on the heap and their reference is stored on the stack.

Integers overflow is checked and the system automatically switches to big integers (allocated on the heap) if necessary.

```
: fact        \ n -- n!
   | i | 1 swap loop: i [ i * ] ;
```

```
500 fact .
```

Integers can be written in hexadecimal or binary using **#0x** and **#0b** words (there is no such words as BASE in Oforth) :

```
0xFFFF0000 .
0b01001111 .
```

In addition to number's operations and Integer operations listed in the Arithmetic chapter, Integer class declares the following words :

```
isEven        \ n -- b : Returns true if n is even
isOdd         \ n -- b : Returns true if n is odd
bitAnd        \ n1 n2 -- n3 : Do a bit and between n and m
bitOr         \ n1 n2 -- n3 : Do a bit or between n and m
bitXor        \ n1 n2 -- n3 : Do a bit xor between n and m
bitLeft       \ nb n -- m : Do a shift of n (nb bits to the left )
bitRight      \ nb n -- m : Do a shift of n (nb bits to the right )
seqEach       \ r n -- : Perform r on integers between 1 and n.


#. 10 seqEach
```

Other operations on integers are declared into the optional math package. To use them, you will have to import this package (see Package chapter) :

```
import: math
```

5.5 Boolean

There is no dedicated type for booleans. Booleans are implemented as integers.

Booleans **true** and **false** are constants respectively equals to **1** and **0** and any object different from **0** is considered as a true value.

Integer class implements those operations for booleans :

```
not                \ b1 -- b2
and                \ b1 b2 -- b
or                 \ b1 b2 -- b
xor                \ b1 b2 -- b
```

Those operations are not performing operations on bits. They return booleans. Operations on bits are declared for Integers (#bitAnd, #bitOx, #bitXor, ...).

Boolean objects are small integers and are not allocated on the heap.

5.6 Character

There is no dedicated type for characters. Characters are implemented as integers, which represent their unicode code.

A char is entered using word **#'** :

```
'a' .
```

Integer class implements those operations for characters :

```
isLwSpace           \ c -- b : Returns true if c is BL or HTAB
isSeparator         \ c -- b : Returns true if c is BL, HTAB or LF
isUpper             \ c -- b : Returns true if c is an upper case char
isLower             \ c -- b : Returns true if c is a lower case char
toUpper             \ c -- v : Returns upper of c
toLower             \ c -- v : Returns lower of c
isDigit             \ c -- b : Returns true if c is a digit
isLetter            \ c -- b : Returns true if c is a letter
isAlpha             \ c -- b : Returns true if c is a digit or a letter

asDigitOfBase        \ base c -- m | null : Returns value of c in base base
asDigit             \ c -- n | null : Returns value of c in base 10
16 'F' asDigitOfBase .
'8' asDigit .

asCharOfBase        \ base n -- c | null : Returns char value of n
asChar              \ n -- c | null : Returns char value of n in base 10
16 15 asCharOfBase .
6 asChar .
```

The word **'** detects some special characters :

```
'\n'                \ New line
'\r'                \ Carriage return
'\t'                \ Horizontal tab
'\b'                \ Backspace
'\"'                \ Double quotation mark
```

```
'\''                \ Single quotation mark
'\\'                \ Backslash
'\uxxxx'            \ Character which unicode code is hex xxxx
'\Uxxxx'            \ Same a \u
```

Characters are small integers and are not allocated on the heap.

5.7 Float

Floats are 64 bits (even on 32bits versions). Floats have the following form :

```
n[.m] or n[.m]e[p]
```

In addition to number's operations and words listed in the Arithmetic chapter, following words are defined in Float class :

```
rand            \ Float -- f : Random number between 0 and 1 excluded
sqrt            \ f1 --  f2
inv             \ f --  1/f
root            \ f n -- f^(1/n)
ln              \ f1 -- f2
log             \ f1 -- f2
exp             \ f1 -- f2
cos             \ f1 -- f2
sin             \ f1 -- f2
tan             \ f1 -- f2
acos            \ f1 -- f2
asin            \ f1 -- f2
atan            \ f1 -- f2
```

Trigonometric methods work with radians.

Some float constants are also defined :

```
E               2.71828182845905
Ln2             0.693147180559945
Ln10            2.30258509299405
Pi              3.14159265358979
PInf            \ Positive infinite
NInf            \ Negative infinite
```

Other operations on floats are declared into the optional math package. To use them, you will

have to import this package (see Package chapter) :

```
import: math
```

On Oforth 32bits, floats are allocated on the heap.

5.8 Block and anonymous functions.

Blocks are anonymous functions.

They are created (at the interpreter level or into a function) using #[and] words.

```
#[ sq 1+ ] .
```

When created, the block is pushed on the stack. As functions, a block is performed using #perform method.

Block can be used to write small pieces of code that we don't want to name as a factor. This is often used for parameters to higher order functions (see the dedicated chapter).

A block can also be used to create closures. It can use locals declared into the function that created it. If so, it copy the locals values at the time the block is created and it becomes a closure ie it can use those values even if the function that created the block has returned (removing its parameters values from the return stack) :

```
: f( n -- aBlock )
    #[ n + ] ;

10 12 f .s perform .s
```

A block or a closure can be declared as the action of a new function using #=>

```
: f( n -- aBlock )
    #[ n + ] ;

2 f => 2+
10 2+ .s
```

Some examples :

```
: compose ( f g -- aBlock )
    #[ g perform f perform ] ;
```

This function returns a block that, when performed, returns the composition of #f with #g.

```
#1+ #[ 2 * ] compose => g
```

```
5 g .s
```

A block can be nested into another block.

```
10 #[ #[ 10 fib ] bench ] times
```

A block can use data on the stack, but it can't declare parameters (perhap's in a future version). I you want to do this, you can declare a local variable into the function that created the block and use it as parameter(s) into the block :

```
: test( f -- aBlock )
   | x | #[ ->x  x 1+ f perform  x f perform  - ] ;


10 #sq test perform .
```

Blocks are alloacted into the dictionary. If a block is a closure, it is allocated on the heap.

5.9 Symbol

A symbol is an identity string : only one version of a symbol exists in the system. Two symbols that have the same value are the same symbol, ie the same object (which is not true for strings).

A symbol is created (or just pushed on the stack if it already exists) using word $

```
$apple $banana
$apple $apple = .s
```

All words name are symbols.

Symbols are often used to defines enumerations. For instance :

```
[ $apple, $banana, $orange ] const: Fruits
```

Symbols are allocated into the dictionary.

6 Object Oriented Programming

6.1 Introduction

Until now, we have talked a lot about objects but we have almost never used OOP mechanisms (at least not explicitly).

This can seem weird, but it is one of Oforth objectives : you can write code without some "heavy" OO concepts and constraints that comes with some other OO languages. You could even write code without knowing a lot about OO programming : you just write your functions and use them. Each basic type comes with its interface and you use them to create your program. At this point, the main difference with classical Forth is that polymorphism allows having the same name for different operations. For instance, #+ is a method that can be used on integers, floats, strings, ... instead of having a different name for those operations.

But, it is now time to talk about OOP and OO meta-model.

OOP is about creating objects that will encapsulate their data. Those objects expose methods that will be used as its interface to use them. When you use a string or a float or big integer, you do not care about the data stored inside those objects : you just use them. You adds two strings, two lists, two big integers and you have a result. This is called **encapsulation**. If the internal storage of a big integer is changed, your program should not be impacted a lot because your program uses a public interface that is be quite stable : you push two big integer on the stack and use #+ to add them.

Encapsulation is implemented by defining classes that will hold attributes and methods exposed. Those classes also allows to create objects (like a factory) and each object will have its own values for the attributes declared for the class. And, it will respond to methods declared into its class.

The second concept is **polymorphism** : a method with the same name can be implemented by different kind of objects, even if objects are quite different : #+ is implemented for floats, strings, arrays, ... Each implementation is quite different, but the method have the same name. The system decides which implementation to use according to the object receiving the method. The object receiving the method is always the top of the stack.

And the last concept is the class **hierarchy** : an integer is a number so all methods defined at the number level can be used by integers and all internal data (attributes) declared at a parent level is also available at a child level.

This concept of hierarchy is the most tricky because it can be very easily misused. And it can lead to the opinion that OO is about describing the "world" with a class hierarchy where each class

should have its place. In Oforth, the class hierarchy is very limited. This is possible because it is not required for two classes to have a common parent to implement the same method : all classes can implement all methods, whatever the hierarchy is and whatever their parents are ("duck typing"). We will come back to this concept of hierarchy later.

6.2 Classes and attributes

Like functions, a class is a word. And like functions, it is also an object of type **Class**. So, creating a new class is asking to the class **Class** to create a new instance :

```
new:          \ aParent Class "name" [ ( mutable "attname" ) ] --


Object Class new: Number
```

This creates the class Number into the dictionary with Object as its parent.

It is possible to define attributes for a class : attributes will be the internal data of a class and each object of this class will have its values for those attributes.

Those attributes are private : they can be accessed only by methods declared for this class :

```
Object Class new: Person( name, age )
```

Objects of type Person will have two data, one name and one age.

#new allows to create an object of a certain class by asking this class to create an new object.

```
new           \ aClass -- aObject
Person new .s
```

Advanced topic : objects allocation.

#new creates object on the heap.

It is also possible to create an object into the dictionary using #allot method on a class.

Objects created from class Person are not very useful because their attributes are initialized to null value and there is no way to update those values. By default, objects are immutable and, after creation, they can't be updated.

In order to define attributes values for immutable objects, we must set them during an object

initialization. After, it is too late...

#new always calls an **#initialize** method with the new allocated object as its receiver. Attributes should be set in this method.

Creating a implementation of #initialize for a class is almost like creating a function, but **#method:** is used instead of #:

```
Person method: initialize  \ s n aPerson --
 := age  := name ;
```

#:= is a word that consumes an object on the stack and store it as the attribute's value of its receiver.

Now that #initialize is defined for a class, we can create a new Person by giving attributes value :

```
"John" 24 Person new .s
```

Now attributes are set during initialization. To see them, we can also provide an implementation for **#<<** method, that is used by #.s to print an object :

```
Person method: <<          \ aStream aPerson -- aStream
 @name << " : " << @age << ;
```

#@ is the word to retrieve an attribute value of the receiver and push it on the stack.

Default behavior for objects is immutability ie that, after initialization, objects can't be updated anymore. Immutability rules are checked at runtime. If you try :

```
Person method: setAge             \ n aPersonn --
 := age ;


"John" 24 Person new 25 over setAge .s
```

An exception is raised because we are trying to update an immutable attribute. If you want to update an attribute after initialization, we must define this attribute as mutable when we create the class, using **mutable** keyword :

```
Object Class new: Person ( name, mutable age )
```

Now the age of a person can be updated after initialization. But the drawback is that objects created from Person are now mutable objects. Mutable objects have restrictions that are also checked at runtime : they cannot be the value of a constant, they cannot be the value of an immutable attribute and they cant be shared between tasks (see concurrent programming).

As classes are objects, they respond to methods declared into the class Class :

```
name                \ cl -- aSymbol : Return class name.
superclass          \ cl -- cl : Return parent class
```

```
isChildOf           \ cl1 cl2 -- b : Return true is cl1 is a parent of cl2
implementor         \ m cl1 -- cl2 : Return implementor of method m for cl1
understand          \ m cl -- b : Return true if class cl understand method m
```

Advanced topic : no class attributes

It is not possible to create class attributes (a value shared by all instances of a class). For instance, a “Singleton” pattern is an anti-pattern in Oforth because this would require synchronization in a multi-threading context.

Everything that could break isolation between tasks (see Concurrent Programming) is not allowed.

6.3 Methods and implementations

Like functions, methods are words that can be executed. Unlike functions, the same method can have multiple implementations, one by class.

The method object itself is not related to a particular class and can be created without defining a particular implementation :

```
Method new: mymethod
```

This creates a new method into the dictionary. To create an implementation for a particular class, **#method:** is used :

```
aClass method: mymethod [ ( param1 param2 ... paramn ) ]
[ | var1 var2 ... varm | ]
 instructions ;
```

If the method object does not exist, it is created by #method: before creating the implementation. That is why creating a method without implementation is seldom used.

Calling a method is different from calling a function. As a method has multiple implementations, the correct implementation to run must be found. To do this, the VM checks, at runtime, the class of the object on top of the stack and it is the implementation corresponding to this object's type that will be chosen. If no implementation is found a "does not understand" exception is raised.

If an implementation is found, its code is launched. Before executing instructions, **the top of the stack is removed from the stack and stored on the return stack** as an implicit parameter of the method : this parameter is called **the receiver** of the method. During the method's instructions, it can be pushed on the data stack using #**self** word.

This is a very important rule : **the receiver is removed from the stack and stored as "self" parameter before beginning the code**.

To implement a dup as a method, self must be pushed on the stack twice.

```
Object method: mydup \ x -- x x
 self dup ;
```

To implement a drop as a method, as the receiver is removed from the stack, there is nothing to do:

```
Object method: mydrop      \ x --
 ;
```

Remembering that the receiver is removed from the stack when you call a method is almost all what you have to know to write methods compared to functions. Compare those implementations that are doing the same thing (calculate inverse hyperbolic cosinus of a float):

```
: acosh1      \ f1 -- f2   : Calculate acosh of f1 : f2 = ln(f1 + sqrt(f1^2 - 1))
   dup sq 1.0 - sqrt  + ln ;


: acosh2( f -- f1 )
   f sq 1.0 - sqrt  f +  ln ;


Float method: acosh3 -- f
   self sq 1.0 - sqrt self + ln ;


1.5 acosh1 .
1.5 acosh2 .
1.5 acosh3 .
```

Like functions, an implementation can defines parameters and local variables.

```
Object method: test( a b -- n )
 self b + a - ;


10 20 30 test .s
30 test ( 10, 20 ) .s
```

Here, the receiver is 30 (the receiver is always the top of the stack) so it is removed from the stack and stored as self parameter. Next, two parameters are declared so a value is 10 and b value is 20.

The “sugar” notation works also for methods but the the receiver must remain on top of stack and is not part of the parameters. For instance :

```
put           \ i x aArrayBuffer  : Put x at index i of aArrayBuffer
```

```
10 1.2 aArrayBuffer put
aArrayBuffer put(10, 1.2)
```

Methods are objects of class Method and, like functions, they can retrieve by # word :

```
Object method: test
 self 1+ ;

#test .s
10 #test perform .s
```

Methods respond to :

```
name          \ m -- aSymbol : Return method name.
perform       \ x m -- : Perform method m with x as its receiver.
compile       \ m -- : Compile a method into the current definition.
```

If you look at the examples in this chapter, you can see that classes are not closed : there is not a beginning and an end for a class definition. You create a class and you can add method implementations for this class whenever you want.

You can also create class methods implementations. A class method is a method for which the receiver is the class itself (and not an instance of the class). This is done using **classMethod:** instead of method:

```
Float classMethod: new      \ Float -- 0.0
 0.0 ;

Float new .s
```

For instance, #new is declared as a class method of Object.

6.4 Polymorphism

Polymorphism is the ability for methods to have different implementations.

Each class, whatever its position into the class hierarchy, can declare an implementation for a method.

```
Object Class new: A
A method: m  "I respond to an A object" . ;
```

```
Object Class new: B
B method: m "I respond to a B object" . ;

B Class new: C

A new m
B new m
C new m
```

It is also possible to overload an implementation into a subclass, but only if the implementation is declared as **virtual** into the parent class :

```
Object Class new: A
A method: m "I respond to an A object " . ;

A Class new: B
B method: m "I respond to a B object " . ;
```

This will raise an compilation error as m is not defined as virtual.

```
Object Class new: A
A virtual: m "I respond to an A object " . ;

A Class new: B
B method: m "I respond to a B object " . ;

A new m
B new m
```

Into an overloaded method, it is possible to call the implementation at the upper level using #**super**. #super is like #self but the implementation called is the one of the superclass :

```
Object Class new: A
A virtual: m "I respond to an A object" . ;

A Class new: B
B method: m  super m  "but it is a B" . ;

A new m
B new m
```

6.5 When no hierarchy is better than bad hierarchy

Oforth implements single inheritance : each class has one and only one parent. So this link is very strict and implement a "IS-A" relation between two classes.

- An Integer IS-A Number
- A Array IS-A Collection.

Each attribute and each method declared at a parent level is available at the child level and, consequently, should be fully applicable at this level. No exception. Beginning to create too much virtual methods at the parent level in order to overload them at the child level can be a sign of a bad hierarchy.

My advice is to create all methods at the parent level as methods and not as virtual :

- Child classes will not be able to override the method implementation and this can be important for security purposes.
- Some calls to this method can be optimized while compiling.

When there is a need to overload a method at a child level, the first question should be : why ? If the relation is really a IS-A relation, methods at the parent level should apply or at least be a part of what the method should do at the child level. Most often, this means that #super should be used at the child level.

Nothing is provided apart from #super to call something at the upper level. This is on purpose. Some code can become unmaintainable otherwise. In a A->B->C hierarchy, a method defined into class C can't call a method at A level if this method is declared at B level.

Class hierarchy is not a simple topic and, if badly used, can generate complex code. Here are some guidelines :

A class hierarchy is not about describing the entire world. It is about describing types the program you are writing will use. Let's take an example : a Car. If you are writing a program to create cars, your Car class will have a lot of data (motor, paint, ...). If you are writing a program to handle traffic jam, your car will have completely different data and methods. It is a non-sense to create a Car class that will work for both programs. Your hierarchy and objects will depend on the program you are writing and are specific to that program. Of course, both programs can use the same tools (collections, network, ...), but, generally, you don't subclass those tools and their hierarchy. Your own hierarchy and objects depend on the program you are writing.

A class hierarchy is not about subclass classes in order to add new attributes, it is about describing a IS-A relation between two types. And this IS-A relation is program dependent. Let's say you have a class Point with two attributes (x and y). Now you want to create a 3D Point (with x, y, z). Will you subclass the Point class and adds a z attribute ? Again, it depends on the program you are writing. If you are writing a general mathematical library, no, a 3D Point is obviously not a 2D Point. Why a 2D point would be a parent of a 3D Point ? But, for the program you are writing and the notion of Point you are trying to describe, it could be perfectly acceptable.

A class hierarchy is not about masking attributes into subclasses, it is about describing a IS-A relation between two types in your program. If you use the same example, some will do the

opposite : they create a 3D Point class and create a 2D Point as a subclass of the 3D Point ie a 2D Point is a 3D Point that does not uses z or set it to 0. This is not good too, because you probably will have to overload many 3D Point methods to update or forbid them at the 2D Point level. If you use "super" into 2D Point method, you will probably have problems to deal with.

The purpose of a class hierarchy is not to deal with those kind of problems. If you spend time on them, probably your hierarchy itself must be changed.

6.6 Properties : when no hierarchy is better than bad hierarchy... again

Here is the problem : objects respond to #== and objects of some classes can be ordered by implementing #<=. With those two methods, it is possible to implement other comparisons :

```
< > >= min max between
```

It would be interesting to code those words just once. How to do this ?

Languages like C++ can handle this using multiple inheritance, but, I think, at the cost of an more complex OO model.

With single inheritance, one way (and the only way on some languages) is to create a Comparable class, to define those methods and to have all classes that implement #<= be child of this class. Then, all objects of those classes will understand all comparisons methods.

So we declare Integer, Float, Date, ... be subclasses of Comparable... But wait... Integer is also a Number. Should Integer be a subclass of Comparable or a subclass of Number ?

Declaring a class to be a subclass of a Comparable class is a bad design because you do not implement a "IS-A" relation, you implement a "IS" relation : being comparable is a property of some objects, not what make them being of a certain type.

To handle this, Oforth meta-model implements properties; properties are words like classes, they can implement methods and can have attributes, but :

- There is no hierarchy between properties
- You can't create objects from properties.
- Classes must have to implement some methods to be of a particular property (here #<= for instance).

The comparable property could be implemented like this (see Comparable.of for the full version) :

```
Property new: Comparable

Comparable requires: <=


Comparable method: >       \ x y -- b
```

```
   self <= not ;
Comparable method: <( c )  \ x y -- b
   self c == not c self <= and ;
Comparable method: >=      \ x y -- b
   self < not ;
Comparable method: min     \ x y -- min(x,y)
   self over <= ifTrue: [ drop self ] ;
Comparable method: max     \ x y -- max(x,y)
   self over <= ifFalse: [ drop self ] ;
Comparable method: between( x y -- b )
   self y <=  x self <=  and ;
```

Now that this property is created, it is possible for classes to be of this property :

```
Integer is: Comparable
Float is: Comparable
Date is: Comparable
```

This will raise an exception if the class does not implement #<=

If you identify a common pattern between various classes and if class hierarchy is not relevant, probably a property is what you are looking for.

Apart from Comparable, two other properties are already defined : Runnable and Indexable. Indexable will described into the chapter about collections. Runnable is a property attached to each class which objects can be performed. #perform is required and methods declared into this property are :

```
under         \ x y r -- z y : Perform r on the second element.
times         \ n r -- ...   : Perform r n times
bench         \ r -- ...     : Perform r and print elapsed time.
```

Functions, methods, operators, blocks, dynamic proc, deferred words, ... are Runnable.

6.7 Polymorphism revisited

With properties, finding the correct implementation for a method must be revisited to include search for methods implemented into the class properties. The final algorithm to find the right implementation of a method is :

1) Find the object's class on top of stack.

2) Find an implementation of the method to run at this class level. If found, run it.

3) Find an implementation into properties declared for this class (beginning by the last one). If found, run it.

4) Find the superclass of this class. If not null, go back to 2).

5) Throw a "does not understand" exception.

6.8 Function or Method ?

When you call a word, there is no syntax difference between calling a function and calling a method. So, sometimes, you will have to make a choice : should I create a function or a method. Here are some guidelines.

Functions and methods are runnable words and they associate a name to a piece of code. A word name is unique into a system (in fact into a package, see Packages chapter for more information).

A function should be created when you want to implement a global feature in your system :

- You will bind code to its name and this can't be changed.
- There is no polymorphism.
- Generally, functions run faster that methods (no polymorphism to handle at runtime).
- Functions are not a feature of a particular class.
- General words like open, close, set, get, new, at, put, +, -, ... verbs, ... should not be created as functions : they can have a different purpose for various classes. If you create a function, you will forbid a method with this name for any class.
- Some private features of a class can be implemented as a functions.

A method should be created when you want to implement a feature for a particular class :

- A method has a receiver and this receiver is this object that will receive the message : the method should be related to this object specifically.
- Polymorphism is allowed : all classes will be able to define code for this method.
- General words liken open, close, set, ... should be created as methods.
- Particular words like "open_file" should not be created as methods : it is redundant to send an open_file method on a file object... The method you are looking for is "open".
- If your method is virtual, subclasses (even not yours) will be able to overload your code.
- If your method is not virtual, you will block subclasses : they won't be able to overload your code.
- Using methods, you will detect some errors sooner : if a method is not declared for an object, you will know it at once when calling the method. With functions, the function will run and the error will raise later, as the object on top of stack will probably not respond to a call into the function (or a function called by a function ... called by the function).

Even with those guidelines, you will sometimes hesitate between a function and a method. For instance, #== is a virtual method at the object level and can be overloaded by classes. What about #<> word ?

It can be a non virtual method (in order to not allow overload) into Object class :

```
Object method: <>    \ x y -- b
 self == not ;
```

Or it can be a function :

```
: <>                 \ x y -- b
 == not ;
```

The choice was made to create a function, but a method would have been a reasonable choice, too.

7 Dictionary and OO meta-model

We have already encountered some word types (Class, Property, Function, ...). In this chapter, we list all word types of the OO meta-model.

The dictionary is an area where all created words are stored. When a name is typed, the interpreter searches for the corresponding word into the dictionary.

7.1 Words

Words created into the dictionary inherit from Word class :

```
 Node
------ Attribute
------ Implement
------ Word
--------- Implementor
------------ Class
------------ Property
--------- Function
--------- Defer
--------- Method
--------- Operator
--------- Constant
--------- TVar
--------- Alias
--------- Package
--------- Struct
--------- OSLib
--------- OSProc
```

A Node is just an object with a "next" attribute, that allows to implement linked list of words (words are stored into the dictionary as hash tables).

Two types are not words : Attribute, that describes one class (or property) attribute and Implement that describes a method implementation for a particular class.

All words have a name and this name is unique into the system (well, into the package). A word

name is a symbol. Word # allows to retrieve a word by name from the dictionary :

```
#               \ "name" -- aWord | null : Read a name and retrieve corresponding word.
```

Methods implemented at the Word level are :

```
find            \ str Word -- aWord | null : Find a word in the dictionary
name            \ aWord -- aSymbol : Returns word's name.
alias:          \ aWord "name" --  : Creates an alias of aWord with "name"
forget          \ aWord --
```

#forget makes a word no more findable in the dictionary (but definitions that use this word will continue to work).

Class, Property, Function, Method and Operator are word types that are already described, so we pass them.

7.2 Constants

A **Constant** is a word that returns a constant value. When a constant name is used (at the interpreter level or into a body), its value is pushed on the stack. To create a constant, the provided value must be immutable.

```
const:          \ x "name" -- : Creates a constant with "name" and value x


2.0 ln const: Ln2
```

7.3 Task variables

A **TVar** is a global variable. When a TVar is created (using tvar:) , its value is null. Using a TVar name pushes the value on the stack. #to allows to modify a TVar value :

```
tvar:           \ "name" -- : Creates a new tvar initialized with null value.


tvar: myvar
myvar .s
2.3 to myvar
myvar .s
```

You should not rely on Tvar too much. One important characteristic your functions or methods should have it to answer the same return when called with the same parameters. This is not sure if you use a TVar. So each time it is possible, it is better to send the values needed by parameters on

the stack.

Sometimes, you will need a global state and a TVar could be used (but you'd better think twice...).

Advanced topic : tvars don't break task isolation

tvar are global words, but each task has its own value (hence its name: a task variable). You can't use a tvar to share values between tasks. This is the same mechanism as USER variables in Forth.

There is no word to create a variable global to the whole system. This would break isolation between tasks.

7.4 Aliases

An **Alias** word is a word created as an alias for another word. It is often used to create aliases of words created into packages, but can also be used to create shortcuts, ... An alias is really an alias : when an alias is searched in the dictionary, it is the original word that is returned.

```
#self alias: this
#this .s
```

7.5 Deferred words

Sometimes, we need to define a function by its name but define its behavior after (or change its behavior by adding features). Deferred words allow to do this. A deferred word is created with #defer:

```
defer:        \ "name" --


defer: myword
```

This creates a deferred word with an empty behavior. If you try to execute a deferred word without defining a behavior, you'll get an exception. But you can use this word into a definition before defining its behavior.

After a deferred word is declared, you can get and set its behavior using #action and #act :

```
#action       \ aDefer -- aFunction
#act          \ aFunction aDefer -
```

```
defer: myword
#myword .s drop
#dup #myword act
#myword .s drop
10 myword .s
#myword action .s drop
#drop #myword act
#myword .s drop
myword .s
```

Updating a deferred word behavior does not only impact future compilation of this word into definitions: all functions that previously compiled this deferred word will have their behavior changed too :

```
defer: myword
#dup #myword act
: test  10 12 myword ;
test .s
#2dup #myword act
test .s
```

The action of a deferred word can also the a block or a closure :

```
defer: myword
#[ 1+ ] #myword act
10 myword .
: test( a -- aBlock)  #[ a + ] ;
10 test #myword act
10 myword .
```

If you want to fix the behavior of a deferred word forever, you can freeze it. If so, #put will raise an exception.

```
freeze        \ aDefer -


#myword freeze
#swap #myword act    \ Raise an exception
```

Like functions, a deferred word can be immediate (see Compilation chapter). But, setting the behavior of a deferred word with an immediate function does not make it an immediate word. You must set the immediate attribute when creating the deferred word :

```
defer: myImmediateWord  immediate
```

Deferred words can be used :

- To define an action that will be updated later
- To define mutual recursion (you can compile a deferred word before defining its behavior).

One difference between classical Forth and Oforth is that, in Oforth, you can't redefine a word. If you want to be able to redefine a word behavior, the only option is to define a deferred word. If you do so, you must be aware that later code can redefine this behavior and the possible security impact. If the deferred word usage is safe, it's ok. Otherwise, you should freeze you deferred word when its behavior is finalized.

7.6 Other words

Package words are described into the Packages chapter

Struct, OSLib and OSProc words are described into the FFI chapter.

8 Compilation

Oforth is an interpreter : it reads words and perform them. Some of those words (#:, #method: , ...), when performed, create new words into the dictionary and associate code to them.

To compile code, there is no separated compilation phase : the interpreter generates native code on the flow. As soon as a definition is closed, native code has been generated and can be executed. There is no interpretation, no bytecode, ...

The code is generated by a one-pass compilation, while the interpreter read names on the input buffer.

This chapter explains how all this works.

8.1 The current definition

The current definition is the definition the interpreter is currently compiling :

- For functions, the current definition is everything between : and ;
- For methods, the current definition is everything between method: and ;

Outside : (or method:) and ; , the current definition does not exists.

8.2 Interpreter state

Interpreter works with a STATE variable. This variable tells the interpreter if it is running in INTERPRET state or in COMPILE state.

When running in INTERPRET state, the interpreter behavior is :

- Runnables (functions, methods, ...) are performed at once.
- Other words and literals (integers and floats) are pushed on the stack.

It is the state of the interpreter when you enter commands.

When running in COMPILE state, the interpreter behavior is :

- Runnables are not performed : they are compiled into the current definition.

- Some special functions are performed even the interpreter is running in COMPILE state. They are called immediate functions because they have an immediate flag set.
- Other words and literals are not pushed on the stack, they are compiled into the current definition : code is added to push them on the stack at runtime.

STATE variable value should not be modified; it can only be modified by specific words :

- #: function and #method: change the STATE value to COMPILE
- #; function changes the STATE value back to INTERPRET.

Of course, #; has the immediate flag set and is performed even if STATE is COMPILE. Otherwise, it would be compiled into the current definition and it would not be possible to go back to INTERPRET state.

This simple (Forth) mechanism allows generating native code for functions and methods : a one pass JIT compilation while executing the immediate methods called into the current definition.

An example :

```
dup *
```

If you type this command, the interpreter it is into INTERPRET state, so :

- It reads a name (dup), it detects a function and performs it at once.
- It reads a name (*), it detects a method and perform it at once.

Now if you write :

```
: square  dup * ;
```

The interpreter begins in INTERPRET state, too :

- It reads a name (:), it detects a function and perform it at once. This function reads a name (square) on the input buffer and creates a function into the dictionary with name "square". Then it changes the STATE value to COMPILE and finishes.
- As "square" name has been "consumed" by #: , the interpreter does not "see" it. It reads the next name ie "dup" , and detects a function. But STATE is now COMPILE so it compiles it into the current definition.
- The interpreter reads a name (*) and compile it into the current definition.
- The interpreter reads a name (;). it detects a function. STATE is COMPILE, but this function has its "immediate" flag set, so, instead of compiling it into the current definition, it performs it at once. This function closes the current definition and set the STATE value back to INTEPRET.

#square function has now been compiled into native code in a one-pass compilation.

This way to compile definitions fits perfectly with RPN notation; you don't have to wait to read an entire instruction before compiling it. Each function or method is compiled into the current definition the moment it is read by the interpreter.

Advanced topic :

STATE variable values are different from some other Forth :

- 0 means INTERPRET state

- > 0 means COMPILE state : value is 1 into a body and incremented by 1 for each nested block into the current definition.

For a block created directly at the interpreter level, value is 1 (unless nested blocks, of course).

8.3 Native code optimizations

While compiling, various optimizations are applied. In particular :

- Functions like #dup, #over, ... have special treatments when compiled.
- The interpreter keeps track of last push on the stack, last drop, ...
- When a method is called, it tries to identify the implementor to use at compile time. If so, it optimize the call without polymorphism.
- ...

This allows to optimize code even if the compilation is done in one pass.

If you know assembler, Oforth comes with a disassembler that allows to see the native code generated. This disassembler uses GCC syntax.

On x86 32bits processors, register usage is :

```
edi                         Data stack pointer
esp                         Return stack pointer
ebp                         Receiver of a method
esi                         Worker.
eax, ebx, ecx, edx   General purpose registers
```

For instance :

```
: sumDigits( n -- m )
 0 while( n ) [ n 10 /mod ->n + ] ;

#sumDigits see
ff:77: 0:6a :        0 : push 0(edi)
```

```
6a: 8:c7: 7 :        3 : push 8
c7: 7: 0: 0 :        5 : mov c0000000, (edi)
8b:4c:24: 4 :        B : mov 4(esp), ecx
81:f9: 0: 0 :        F : cmp c0000000, ecx
 f:84:2c: 0 :       15 : je 47
8b:4c:24: 4 :       1B : mov 4(esp), ecx
89:4f:fc:bd :       1F : mov ecx, -4(edi)
bd: a: 0: 0 :       22 : mov c000000a, ebp
83:ef: 4:e8 :       27 : sub 4, edi
e8:29:36:27 :       2A : call 401F10
8b: f:83:c7 :       2F : mov (edi), ecx
83:c7: 4:89 :       31 : add 4, edi
89:4c:24: 4 :       34 : mov ecx, 4(esp)
bb:54:39:19 :       38 : mov 193954, ebx
e8:46:2b:27 :       3D : call 401440
e9:c4:ff:ff :       42 : jmp B
83:c4: 8:c3 :       47 : add 8, esp
c3: 0: 0: 0 :       4A : ret
```

This code has been generated on the flow while the interpreter executed immediate functions. In this example, immediate functions executed are (in order) :

```
#( #while #( #) #[ #-> #] and #;
```

8.4 User defined immediate functions

There are various immediate functions that generate code. You can also create your own immediate functions. For this, the #immediate function must be called right after a function definition :

```
: test   "Hi, I'am here" .cr ; immediate
: test1  12 13 test + test ;
```

As #test is immediate, it is performed during compilation by the interpreter and nothing related to #test is compiled into #test1. Test1 was compiled as if it was :

```
: test1 12 13 + ;
```

8.5 Interpreter revisited

With what we know about STATE and immediate functions, we can now describe how the interpreter works :

1) It reads a name on the current input buffer

2) It tries to find a word corresponding to this name into the dictionary.

3) If a word if found and not runnable, it pushes it on stack or compiles a push according to the STATE value and go back to 1)

4) If a word if found and runnable, it pushes it on the stack, call #interpret (see below) and go back to 1).

5) If the name is not a word, the interpreter tries to detect an integer or a float. If so, it pushes it on the stack (of compile a push if COMPILE) and go back to 1).

6) An exception is raised : "Can't evaluate word".

Functions, methods, ... understand #interpret. This method interprets the word according to current STATE value (if interested, you can see #interpret code for each type into the prelude/prelude.of file). For instance, for functions, #interpret is :

1) If STATE is RUNTIME or the function is immediate, #perform is called.
2) Otherwise, #compile is called.

#compile (which can also be found into prelude/prelude.of) compiles the receiver into the current definition.

It is the text interpreter that, along with immediate functions, controls interpretation and compilation of definitions, based on the STATE value.

From the text interpreter perspective, compilation is not separated from runtime; everything is done by executing functions. It just happens that some functions, when executed, generate native code into the current definition...

8.6 The Control Stack

Another stack ? Yes, another stack...

This stack is used during compilation to save information that will permit to resolve jumps, loops, ... As the process is a one pass compilation, everything must be handled during this pass. Let's see an example with the #ifTrue: word .

#ifTrue: is, of course, an immediate function that will be performed by the interpreter during compilation. It :

- Generates code to remove the top of the stack and test this value with false.
- Generates an "empty" conditional jump.
- Pushes the current code address and #then function on the control stack.

"empty" jump means that, at this point, the address where to jump is not already known (but the space for this address it allocated).

After a short time (or a long time...), the end of the #ifTrue: block will be reached, with #]. This word is also an immediate word. Its code is very simple : it removes one object from the control stack and perform it. As #ifTrue: pushed the #then function on the control stack, #then is performed. This function retrieves an address on the control stack, calculate the jump value and update the "empty" jump with the correct value.

Now, our #ifTrue: test is complete and ready to by performed. All other syntax (loops, while, doWhile, continue, break, ...) work exactly the same way. When everything is resolved, the body has been compiled in a one-pass compilation.

As the control stack is a stack, this allows nesting various syntax : tests, loops, ... without limitation on the number of level of nested instructions.

8.7 Directives

Into source files, it is possible to use some directives to condition some parts of the files. Those directives are uppercase to differentiate them from words used into definitions :

```
IFTRUE:              \ b -- : Execute following block if b is non false
System.ISWIN IFTRUE: [ : test "Windows system" .cr ; ]


IFFALSE:             \ b -- : Execute following block if b is false
DEFINED:             \ "name" -- b : Returns true if "name" is a defined word.
```

9 Higher order functions and collections

A higher order function is a function (or method) that takes a runnable (a function, a method, a block, ...) as parameter and/or returns a block as its result.

We have already encountered a higher order function :

```
: compose ( f g -- aBlock )
    #[ g perform f perform ] ;
```

This function takes two runnables as parameters and returns a block.

HOF allows to apply a runnable on collection, to create new collections, ...

9.1 #forEachNext method and #forEach: function

The #forEachNext method is a virtual method. It has a default implementation at the Object level. It allows to traverse items contained into an object in a generic way. Its stack effect is a little complicated :

```
forEachNext          \ x o -- y item true | false
```

o is the object we want to traverse and x is an object that allows to retrieve the next object into o.

This function should retrieve the next item into object o, using x value :

- If there is no more objects, this function just returns false.
- If x is null, this is the first time that forEachNext if called, so the first item is to be retrieved.
- If another item is found, this function should return y, this item and true. y is the object that will be sent back to the next call to #forEachNext as x parameter to retrieve another item.

So #forEachNext is an iterator : each call to #forEachNext retrieve and returns the next item of an object.

At the Object level, #forEachNext is :

```
Object virtual: forEachNext      \ x obj -- y o true | false
   ifNull: [ 1 self true return ] false ;
```

The first time #forEachNext is called, it returns the object itself. The next time, it returns false. So traversing an object, by default, is just returning this object one time.

For null object, nothing is returned, #forEachNext returns false the first time it is called :

```
Null method: forEachNext  drop false ;
```

For Indexables (containers which items are accessed with an index), #forEachNext is :

```
Indexable virtual: forEachNext          \ i coll -- i+1 x true | false
   dup ifNull: [ drop 0 ]
   1+ dup self size <= ifTrue: [ dup self at true return ]
   drop false ;
```

So what is the interest of this method ? It is used by the #forEach: function; this immediate function creates a loop to traverse all items included into an object :

```
x forEach: o [ instructions ]
```

o must be a declared as a local variable. #forEach: removes x from the stack and generates a loop that calls #forEachNext on x; instructions will be executed for each item into x. Into instructions, o value is the value of the current item.

```
: test        \ --
   | o | [ 1, 2, 3, 4, 5 ] forEach: o [ o . ] ;
```

#forEach: is the base instruction for all HOF. As# forEach: uses forEachNext, new collections just need to overload #forEachNext (when needed) to answer to all defined HOF.

9.2 Arrays

Collections are containers for items. Oforth provides various collections. The most used one is an array : an array is an immutable container for items, accessed by an index.

An array can be explicitly created using #[, #, and #] words. As these words are detected by the interpreter (see above), no space is needed before or after them :

```
[ 1, 2, 3, 4, 5 ] .s
```

Items into an array don't have to be of the same type. And they can be arrays too :

```
[ 1, 2.3, "abcd", 'a', [ 1, 2, 3 ]] .s
```

Arrays can also be created using #arrayWith function :

```
arrayWith            \ xi n -- aArray : Create an array of n elements on the stack


12 13 14 3 arrayWith \ Returns [ 12, 13, 14 ]
```

Arrays are also created as the result of executing HOF.

9.3 Higher Order Functions

Object Class implements many HOF. These functions (or methods) have a collection as receiver and a runnable as parameter. The most basic one is **#apply** : it takes a runnable and a object on the stack. For each item into this object, it pushes it on the stack, then perform the runnable on it.

```
apply          \ r x -- ... : Apply r on each item of x


#. 10 apply
#. [ 1, 2, 3, 4, 5 ] apply
0 #+ [ 1, 2, 3, 4, 5 ] apply .s
0 #[ sqrt + ] [ 1, 2, 3, 4, 5 ] apply .s
0 [ 1, 2, 3, 4, 5 ] apply( #[ sqrt + ] ) .s
```

#apply is implemented using #forEach (and therefore #forEachNext). Its code is simple :

```
Object method: apply ( r -- ... )
   | o | self forEach: o [ o r perform ] ;
```

As the object and the runnable are removed from the data stack (they are stored on the return stack as self and r parameter), it is ok for the runnable to use objects on the stack at the moment #apply is called :

```
0 #[ sqrt + ] [ 1, 2, 3, 4, 5 ] apply .s
```

The #+ will accumulate results using the 0 on the stack.

This is a general rule : all HOF have been written to allow the runnables to access objects on the data stack when they are performed.

#applyIf applies a runnable only on items that respond true to a condition.

```
applyIf        \ rcond r x -- ...


0 #isEven #+ [ 1, 2, 3, 4, 5 ] applyIf .s
0 [ 1, 2, 3, 4, 5 ] applyIf(#isEven, #+) .s
```

#reduce is like #apply, but the first items is pushed on the stack as value for accumulator before looping across items :

```
reduce         \ r x --
```

```
#+ [ 1, 2, 3, 4, 5 ] reduce .s
```

#reduceWith is a generalized version of reduce. Before applying the runnable, another runnable is applied on each item :

```
reduceWith          \ p r x -- : reduce x using r, but after applying p on each item


#sq #+ [ 1, 2, 3, 4, 5 ] reduceWith            \ Returns 55
[ 1, 2, 3, 4, 5 ] reduceWith( #sq, #+ )  \ Returns 55
```

#include checks if a element is included into a collection (using #==) :

```
include       \ e x -- b : Returns true if e is included into x
```

#conform checks if all items respond true to a condition :

```
conform       \ rcond x -- b


[ 1, 2, 3, 4 ] conform( #isEven )       \ Returns false
```

#2apply works on 2 collections : it pushes items of the 2 collections, then perform a runnable. The loop stops when one of the collections has no more items.

```
2apply        \ y r x -- ...


0 [ 1, 2, 3 ] #[ * + ] [ 4, 5, 6 ] 2apply            \ Returns 1*4 + 2*5 + 3*6
0 [ 1, 2, 3 ] [ 4, 5, 6 ] 2apply( #[ * + ] )   \ Same...
```

#maxFor returns item that respond the max value for a runnable :

```
maxFor        \ r x -- item


[ [ 1, 2 ], [ 1, 3 ], [ 2, 4 ] ] maxFor( #second ) .s
```

#minFor returns item that respond the min value for a runnable :

```
minFor        \ r x -- item


[ [ 1, 2 ], [ 1, 3 ], [ 2, 4 ] ] minFor( #second ) .s
```

#iapply works only on Indexable collections (see Indexable propertie). It allows to loop with the index : for each item, it pushes the item, its index, then call a runnable

```
iapply        \ r x --


0 [ 2, 3, 4, 2, 3 ] iapply( #[ isEven ifTrue: [ + ] else: [ drop ] ] )
\ Return sum of items at even index
```

9.4 Indexable collections

Some collections are indexable, ie have the propertie to be Indexable. Items can be accessed with an index. First item is at index 1.

#at method is required for a collection to be indexable :

```
at                  \ u x -- item : Returns item at index i, null if none.
```

Apart from Hash, all other defined collections are indexable.

Indexable implements #forEachNext using #at and #size, so this method have not to be redefined (but it can be for performance purpose).

Indexable propertie adds those methods :

```
isIndexable    \ x -- b : Returns true is x is indexable
iapply         \ r x -- : Apply r on item and index.
first          \ x -- y : Returns first item of x, null if none.
second         \ x -- y : Returns second item of x, null if none.
third          \ x -- y : Returns third item of x, null if none.
last           \ x -- y : Returns last item of x, null if none.
indexOfFromTo  \ e i j x -- u : Returns index of e between i and j range
indexOfFrom    \ e i x -- u : Returns index of e beginning at i index
indexOf        \ e x -- u : Returns index of e into x
lastIndexOfFromTo\ e i j x -- u : Returns last index of e between i and j range.
lastIndexOf    \ e x -- u
isAllAt        \ y n x -- b : Returns true if x is all at index n of y
indexOfAllFrom       \ y n x -- u : Returns index of y into x beginning at n
indexOfAll           \ y x -- u : Returns index of all items of y into x
```

9.5 Mapping collections and ArrayBuffers

ArrayBuffer are mutable arrays : you can add or remove items from a ArrayBuffer. ArrayBuffers reallocate memory if needed. So, for optimization purposes, it is possible to create a new ArrayBuffer with a predefined allocation.

```
newSize        \ u ArrayBuffer -- aArrayBuffer : Create an empty ArrayBuffer.
new            \ ArrayBuffer -- aArrayBuffer   : Create an empty ArrayBuffer
newWith        \ n x ArrayBuffer -- aArrayBuffer : Create initialized LB with n x
```

```
addAll        \ x aArrayBuffer --     : Adds all items of x into the LB
asArrayBuffer \ x -- aArrayBuffer          : Creates a LB with items of x
asArray       \ aArrayBuffer -- aArray : Create a Array from a LB.
add           \ x aArrayBuffer --     : Adds x to aArrayBuffer
put           \ i x aArrayBuffer -- : Put x at index i of aArrayBuffer
removeAt      \ i aArrayBuffer -- x : Remove and return item at index x
removeFirst   \ aArrayBuffer -- x   : Remove and return first item
removeLast    \ aArrayBuffer -- a   : Remove and return last item
empty         \ aArrayBuffer --     : Empty a ArrayBuffer
swapValues    \ i j aArrayBuffer -- : Swap values at i and j into aLB
sortElemWith  \ r aArrayBuffer --   : Sort into aLB using sort method r
freeze        \ aLB --            : Mutate a ArrayBuffer into a Array.
```

9.6 Mapping collections

Mapping allows to create new collections from other collections. The collection created depends on the initial collection :

- For string, a string is created
- For all other collections, a Array is created.

Mapping is done by creating ArrayBuffers. Here are the list of methods that map collections :

#map applies a runnable (or a list of runnables) on items of a collection and returns the list of results :

```
map                 \ r x -- aArray

#sq [ 1, 2, 3, 4, 5 ] map         \ Returns [ 1, 4, 9, 16, 25 ]
[ #sq, #1+ ] [ 1, 2, 3 ] map      \ Returns [ [ 1, 2 ], [ 4, 3 ], [ 9, 4 ] ]
```

#mapIf works like #map but collect results only for items that respond true to a condition :

```
mapIf         \ rcond r x -- aArray

#isEven #sq [ 1, 2, 3, 4, 5 ] mapIf     \ Returns [ 4, 16 ]
[ 1, 2, 3, 4, 5 ] mapIf(#isEven, #sq)   \ Returns [ 4, 16 ]
```

#mut is lile #map but works only on ArrayBuffers and updates items with a runnable instead of

creating a new Array. The updated ArrayBuffer is returned :

```
mut                 \ r aArrayBuffer -- aArrayBuffer
```

#filter returns a new collection with only items that respond true to a condition :

```
filter        \ rcond x -- aArray

#[ 3 <= ] [ 1, 2, 3, 4, 5 ] filter      \ Returns  [ 1, 2, 3 ]
"acbDEfgHI" filter( #isUpper )          \ Returns "DEHI"
```

#+ is declared for collections and returns a new collection with the concatenation of all items

```
+             \ x y -- aArray

[ 1, 2, 3 ] [ 4, 5, 6 ] +              \ Returns [ 1, 2, 3 , 4, 5, 6 ]
"abc" "def" +                          \ Returns "abcdef"
```

#- is declared for collections and removes all items of a collection from another collection :

```
-             \ x y -- aArray : Removes items of y from x

[ 1, 2, 3, 4, 5, 5 ] [ 2, 5 ] -        \ Returns [ 1, 3, 4 ]
"abcdefgABCabcd" "bcd" -                \ Returns "aefgABCa"
```

#reverse returns a new collection with elements reversed :

```
reverse       \ x -- y

[ 1, 2, 3, 4, 5 ] reverse              \ Returns [ 5, 4, 3, 2, 1 ]
"abcde" reverse                        \ Returns "edcba"
```

#expand expands all elements into a collection

```
expand        \ x -- y

[[[ 1, 2],[ 4,5 ]], 12, [ 13,14 ]] expand      \ Returns [1,2,4,5,12,13,14]
```

#transpose transpose a Array of subArrays :

```
transpose     \ [[x]] -- [[y]]

[[ 1, 4 ], [ 2, 5 ], [ 3, 6]] transpose   \  [[ 1, 2, 3 ], [ 4, 5, 6 ]]
```

#groupBy groups items according to value they returns with applying a runnable :

```
groupBy       \ r x -- [ [ v, y ] ]
[ 1, 2, 3, 4, 5 ] groupBy(#isEven) -> [ [ 0, [1, 3, 5] ], [ 1, [2, 4] ] ]
```

Some mapping are only available for Indexable collections :

```
groupWith            \ r x -- aArray
Group all adjacent items that respond the same value when r is applied

group          \ x -- aArray     : Group all adjacent elements.
splitBy        \ n x -- aArray   : Split a collection by sublists of n items
sortWith       \ r x -- aArray   : Sort xaccording to method r
sortBy         \ r x -- aArray   : Sort x after applying r on items (with <=)
sort           \ x -- aArray )   : Sort x with method #<=
extract        \ i j x -- aArray : Extract items from index i to index j
sub            \ i n x -- aArray : Extract n items from index i
left           \ n x -- aArray   : Extract the first n items
right          \ n x -- aArray   : Extract the last n items.
tail           \ x -- aArray     : Returns all items but first.
del            \ i j x -- aArray : Return items into x not between i and j index

replaceAll     \ src target x -- y : Replaces all occurences of src by target into x
"abc" "AAA" "abcdefabcghi" replaceAll

zipWith        \ x r y -- aArray : Return Array of results : itemx r itemy
[ 1, 2, 3] [4, 5, 6] zipWith(#+)  \ Returns [ 5, 7, 9]

zip            \ x y -- aArray   : Return Array of pairs [ itemx, itemy ]
[ 1, 2, 3 ] [ 4, 5, 6 ] zip       \ Returns [ [1, 4], [ 2, 5], [ 3, 6] ]

zipAll         \ xi n -- aArray : Transpose n Arrays on the stack.
[1, 2, 3] [4, 5, 6] [7, 8, 9] zipAll(3)   \ [[1, 4, 7], [2, 5, 8], [3, 6, 9]]
```

10 Collection classes

This chapter describes all collections available (apart from Array and ArrayBuffer that are already described). All collections inherit from the Collection class. Collections implemented at startup are:

```
Object
--- Collection
------ Interval            Range of items ( from x to y with step s )
------ Pair                Array with 2 items [ x, y ]
------ Array               General immutable Array of item.
---------- ArrayBuffer     General mutable Array
------------- Stack         Stack with push and pop operations
---------- Json            Json object
------ Hash                Dictionary
------ Buffer              Collection of bytes
--------- MemBuffer        Mutable collection of bytes
--------- String           UTF8 strings
------------ StringBuffer   Mutable strings
```

As all those collections implement #forEachNext, #forEach syntax can be used for all of them, and, therefore, they responds to all higher order functions.

Methods implemented at the Collection level are :

```
isCollection        \ x -- b : Returns true is x is a collection.
size                \ x -- u : Returns nb of items. Must be redefined.
isEmpty             \ x -- b : Returns true if the collection is empty
notEmpty            \ x -- b : Returns true is the collection is not empty.
sum                 \ x -- y : Returns sum of items into x
charsAsString       \ [ c ] -- s : Converts a Array of chars into a string
<<                  \ aStream x -- aStream : Send x into aStream.
```

10.1 Pair

A pair is an Array with 2 elements. They are used as elements of dictionaries [key, value]

```
[ 1, 2 ] .s
```

Because there are 2 elements, a pair is created instead of an Array.

Like Arrays, pairs are immutable and indexable.

If a collection is a collection of pairs, it is possible to retrieve the pair with a key or a value :

```
keyAt          \ x y -- aPair : Returns pair with key x into y
[ [ $a, 1], [ $b, "abc"], [ $c, [1, 2, 3] ] ]  keyAt($b) \ Returns [b, abc]


valueAt        \ x y -- z : Returns value of pair with key x into y
[ [ $a, 1], [ $b, "abc"], [ $c, [1, 2, 3] ] ]  valueAt($b) \ Returns "abc"
```

10.2 Hash

A Hash is a mutable dictionary : each value is associated to a key and items are retrieved and updated using a key.

```
newSize        \ n Hash -- aHash : Returns a hash with n as hash size
new            \ Hash -- aHash   : Returns a new hash with 107 as hash size
```

Methods on hash are :

```
size           \ aHash -- n       : Returns number of elements into the hash
keyAt          \ x aHash -- aNode : Returns node which key is x
valueAt        \ x aHash -- y     : Returns value corresponding to key x


insert         \ key value aHash -- : Insert value at key into the hash.
```

If the key already exists, the value is updated.

10.3 Interval

Interval implements a range of values. Intervals are indexables. An interval is created with :

- The initial value
- The final value

- The step between values.

Methods/ functions of Interval class are :

```
new             \ init end step Interval -- aInterval

size            \ aInterval -- size : Returns number of values.
at              \ n aInterval -- x : Returns value at position n
seqFrom         \ n m -- aInterval : New interval between n and m (step is 1)
seq             \ n -- aInterval   : New interval between 1 and n (step is 1)

10 seq .s
```

Interval are also used to implement #step: loop :

```
n m step step: o [ instructions ]
```

Into instructions, o takes all values between n and m with step.

```
: test          \ --
 | i | 1.0 2.0 0.1 step: i [ i . ] ;
```

10.4 Stack

A stack is a ArrayBuffer subclass that implement **LIFO** stacks. It is a mutable collection.

In addition to ArrayBuffers methods, a stack defines those methods :

```
push            \ x aStack --         : Push x on the stack
pop             \ aStack -- x | null : Pop last item from the stack.
top             \ astack -- x        : Return the last item, but does not remove it
```

10.5 Json

Json is a Array's subclass. It implements JSON objects.

In Oforth, Json objects are built-in and #{, #: and #} words allow to create jsons.

```
{               \ --          : Beginning of JSON object
:               \ --          : Member separator
}               \ -- aJson    : Creates the JSON.
```

```
{ "abcd" : 12, "cde" : { $f : [ 1, 2, 3 ], $g : null }, "fgh" : 1.2 } .s
```

As those words can parse JSON, a string containing a JSON can by parsed as a JSON :

```
{ "abcd" : 12, "cde" : { $f : [ 1, 2, 3 ], $g : null }, "fgh" : 1.2 }
dup asString .s perform .s == .s
```

A ArrayBuffer with pairs of [key, values] can be converted as a json

```
freezeJson     \ aArrayBuffer --

ArrayBuffer new [ "abcd", 12 ] over add [ "cde", [ 1, 2, 3 ] ] over add
dup freezeJson .s
```

It is also possible to create a Json from pairs on the stack using **#jsonWith** word :

```
jsonWith       \ pair*i n -- Json : Creates a json with n pairs.
```

10.6 Buffer

A buffer is a collection of bytes.

Buffers are indexable but, because subclasses can differentiate number of bytes and numbers of characters (UTF8 characters for instance), it is necessary to have separated methods :

```
basicSize      \ aBuffer -- n   : Returns number of bytes
size           \ aBuffer -- n   : For some classes, returns number of chars
basicAt        \ i aBuffer -- n : Return byte at position i
at             \ i aBuffer -- n : Return character at position i (unicode)
```

For Buffer and MemBuffer, #at returns the same value than #basicAt

To handle UTF8 conversions, utf8At is declared :

```
utf8At           \ i aBuffer -- n | null : Return utf8 unicode at i position
asStringifError \ b aBuffer -- aString : Translate a buffer int a UFT8 string.
asString         \ aBuffer -- aString   : Translate a buffer int a UFT8 string.
```

#asStringIfError translate tries to translate a buffer into an UFT8 string. If an error occurs while decoding UTF8 characters, either an exception is raised (if b is true) or character is replaced by '?' (if b is false). #asString replces characters with '?' if error.

10.7 MemBuffer

MemBuffer is a Buffer subclass and implements memory buffer with Read/Write access.

In addition to Buffer methods, MemBuffer adds :

```
newWith       \ n byte MemBuffer -- aMemBuffer
Initialize a MemBuffer of size n with byte.

basicPut      \ i byte aMemBuffer -- : Put byte at i position
put           \ i byte aMemBuffer -- : Same as basicPut
```

For those two methods, if i index is out of the MemBuffer range, an exception is raised.

10.8 String

Strings are a collection of UTF8 characters. String is a subclass of Buffer and are Comparable and Indexable. String characters are accessed using #at method. In order to retrieve a byte from a string, you can use #basicAt (declared into the Buffer class) :

```
at                \ i s -- c   : Returns UTF8 character at position i (1 based)
basicAt           \ i s -- n   : Returns byte at position i (1 based).
```

Word **#"** allows to create new constant strings. As " is detected by the interpreter, no space is required after it.

```
"Hello world!" .s
```

The world #" detects special characters :

```
\n                \ New line
\r                \ Carriage return
\t                \ Horizontal tab
\b                \ Backspace
\"                \ Double quotation mark
\'                \ Single quotation mark
\\                \ Backslash
\uxxxx            \ Character which uncode code is hex xxxx
\U                \ Same a \u
```

Words defined for strings are :

```
size              \ s -- n          Returns number of UTF8 chars
at                \ i s -- c        Returns UTF8 character at index i or null
```

```
==                  \ s1 s2 -- b       Return true is s1 and s2 have the same value
<=                  \ s1 s2 -- b       Returns true if s1 <= s2
hashValue           \ s -- n           Returns hash value of s
perform             \ s --             Perform the string.
eval                \ s --             Evaluate the string.
load                \ s --             Load file which name is s.
asIntegerOfBase     \ base s -- n      Return integer value of s into base
asInteger           \ s -- n | null    Return integer value of s into base 10
asFloat             \ s -- f | null    Return float value of s.
asNumber            \ s -- n | f | null Return integer or float value of s
asSymbol            \ s -- aSymbol     Returns symbol corresponding to s
toUpper             \ s1 -- s2
toLower             \ s1 -- s2
asStringBuffer      \ s -- sb          Creates a new stringBuffer with string value.
+                   \ s1 s2 -- s3      Adds two strings.
sort                \ s1 -- s2         Sort s1 and returns a new string

extractAndStrip     \ i j s1 -- s2
Return substring between i and j after removing leading and trailing spaces.

strip               \ s1 -- s2
Return s1 after removing leading and trailing spaces.

<<wj                \ aStream w j s -- aStream      Send formatted string
<<w                 \ aStream w s -- aStream        Send formatted string

wordsWith           \ c s -- [ s ] : Returns Array of substrings separated by c
';' "abc;def;;ghi" wordsWith .s

words               \ s -- [ s ] : Return Array of substrings separated by space
```

Methods defined for collections allow to handle the inverse operation : create a new string with a collection of strings and a separator :

```
unwordsWith         \ c [ s ] -- s : Returns the concatenation of strings

unwords             \ [ s ] -- s : Returns concatenation separated by space.
[ "abcd", "efg", "hij" ] unwords .s
```

10.9 StringBuffer

StringBuffer (SB) is a subclass of String. It implements mutable strings.
Following words are used to create a StringBuffer :

```
new           \ StringBuffer -- aSB     : Creates a new stringBuffer
newSize       \ n StringBuffer -- aSB   : Creates a new SB and allocates n byte

newWith       \ n c StringBuffer -- aStringBuffer
Creates a SB initialized with n characters c

init          \ n r StringBuffer -- aStringBuffer
Creates a SB initialized with the result of r applied n times.

10 #[ 26 rand 'A' + 1- ] StringBuffer init .s
```

Methods implemented by stringBuffers are :

```
empty         \ aSB --       : Empty a stringBuffer.
addChar       \ c aSB --     : Adds c to the stringBuffer
add           \ c aSB --     : Same as addChar
addAll        \ x aSB --     : Add all elements of x into a stringBuffer
put           \ i c aSB --   : Put c at position i

freeze       \ aStringBuffer --         : Mutate a stringbuffer into a string
asString     \ aStringBuffer -- aString : Creates a new string with stringbuffer value
```

11 I/O and formatting

Input/Output are handled by files and console (sockets are described into the tcp package).

A stream is an object that can receive objects after formatting them. Streams are :

- Files
- StringBuffers

11.1 Formatting objects

Formatting objects is the action to send objects in a particular format to a stream. A stream can be a File or a StringBuffer. The same words are used to format objects, whatever the stream is. The first word is #<< :

```
<<            \ aStream x -- aStream    Sends object x to a stream
<<c           \ aStream c -- aStream    Sends character c to a stream
```

This function sends a basic formatted version of x into a stream. This function is used by #.s to print the stack. #<< leaves the stream on the stack in order to use consecutive calls :

```
System.Out "aaaa" << 12 << Integer << 1.3 << drop
StringBuffer new "aaaa" << 12 << Integer << 1.3 << .s
```

To define a specific format, other functions are available.

#<<w allows to define a width to format object. If the formatted output of the object is greater than this width, the parameter is ignored. Otherwise, the object will be formatted using this width with a default justification

```
<<w           \ aStream w x -- aStream


System.Out "abcd" <<w(8) "cdef" <<w(8) 1.3 <<w(8) 12 <<w(5)
```

#<<wj allows changing the default value to justify the object :

```
<<wj          \ aStream w justif x -- aStream


System.Out "abcd" <<wj(10, JUSTIFY_RIGHT)      \ Output "      abcd"
System.Out 12 <<wj(10, JUSTIFY_LEFT)           \ Output "12        "
```

#<<wjp allows to define a precision for the output format. This method is only available for numbers (Integers and floats) :

```
<<wjp          \ aStream w justif precision aNumber -- aStream


System.Out  12 <<wjp(10, JUSTIFY_RIGHT, 5)     \ Output "     00012"
System.Out -12 <<wjp(10, JUSTIFY_RIGHT, 5)     \ Output "    -00012"
System.Out 1.2234 <<wjp(10, JUSTIFY_RIGHT, 2)  \ Output "       1.2"
```

These methods use #addFloatFormat, #addIntegerFormat, ... that must be defined for the stream used.

As an example, here is how a date is formatted :

```
Date virtual: <<
   self year   <<wjp(0, JUSTIFY_LEFT, 4) '-' <<c
   self month  <<wjp(0, JUSTIFY_LEFT, 2) '-' <<c
   self day    <<wjp(0, JUSTIFY_LEFT, 2) BL  <<c
   self hour   <<wjp(0, JUSTIFY_LEFT, 2) ':' <<c
   self minute <<wjp(0, JUSTIFY_LEFT, 2) ':' <<c
   self second <<wjp(0, JUSTIFY_LEFT, 2) ',' <<c
   self microSecond 1000 / <<wjp(0, JUSTIFY_LEFT, 3)
;
```

There is another method to output a date as a reference date :

```
Date virtual: <<ref
   Date.Days at( self dayOfWeek 1+ ) << ", " <<
   @dd <<wjp(0, JUSTIFY_LEFT, 2) BL  <<c
   @mm Date.Months at << BL <<c
   @yy <<wjp(0, JUSTIFY_LEFT, 4) BL  <<c
   @hh <<wjp(0, JUSTIFY_LEFT, 2) ':' <<c
   @mi <<wjp(0, JUSTIFY_LEFT, 2) ':' <<c
   @ss <<wjp(0, JUSTIFY_LEFT, 2)
;
```

11.2 Files

File objects represent OS files. They allow to read and write into the corresponding system files.

#newMode is used to create a new file :

```
newMode             \ filename mode File -- aFile
   filename is a string corresponding to system file name
   mode is :
        File.BINARY : open a binary file
        File.TEXT : open a text file
        File.UTF8 : open a file containing UTF8 characters.

"myfile" File.BINARY File newMode
File newMode( "myfile", File.BINARY )
```

To create a file with mode = File.UT8, #new can be used :

```
new                 \ filename File -- aFile

"myFile.txt" File new
"myFile.txt" File new
```

Creating a file does not open it. Some methods don't require the file to be open :

```
name          \ aFile -- s : returns file name

stats         \ aFile -- nc nm ns | null null null : Returns file stats
 nc is the number of microseconds for file creation
 nm is the number of microseconds for file modification
 ns is the file size
```

Those values are null if the file is not accessible or does not exists. They can be retrived directly using :

```
exists        \ aFile -- b
size          \ aFile -- ns
created       \ aFile -- nc
modified      \ aFile -- nm
```

In order to read or write, the file must be open :

```
open          \ access aFile --
isOpen        \ aFile -- b : Returns true is the file is open


aFile open(File.READ)
File.WRITE aFile open
```

#open opens a file with an access mode. This method throws an exception if the file can't be open. Access can be :

- File.READ : Open for reading
- File.WRITE : Create and open for writing
- File.APPEND : Open for writing at the end of the file.

Once the file is open :

```
close         \ aFile --          Closes the file. Do nothing if already closed.
position      \ aFile -- n        Returns current file position
setPosition   \ origin offset aFile --  Set file posiition.
```

You can get the current file position using #position and, with some constraints set the file position using #setPosition. #setPosition will set the position according to an origin and an offset (the new position will be origin + offset).

- If the file is created with File.BINARY mode, origin can be File.BEGIN, File.CURRENT or File.END and offset is a number of bytes from this origin (it can be the value returned by a previous call to #position).
- If the file is created with File.TEXT or File.UTF8 mode, origin can only be File.BEGIN and offset can only be zero or a position returned by a previous call to #position.

Methods used to write into a file are :

```
add           \ n aFile --        Adds n to aFile
addChar       \ n aFile --        Same as #add
flush         \ aFile --          Flush pending data.
```

#add writes n to a file :

- If the file is not created as an UTF8 file, the byte n is written.
- If the file is created as an UTF8 file, the byte(s) corresponding to the UFT8 sequence of unicode code n is written. An exception is raised if n is not an unicode code.

Files are buffered and an effective write on disk will occurs when the buffer is full. #flush allows to flush immediately all pending data to write. #flush can't be used on files opened with File.READ (to flush standard input, use #flush on the console, see Console chapter).

A file is a stream, so formatting methods can be used to write to a file (see Formatting objects chapter for details on those methods) :

```
<<c           \ aFile c -- aFile              Write character c ot byte c to a file.
```

```
<<            \ aFile x -- aFile              Write object x to a file.
<<w           \ aFile width x -- aFile
<<wj          \ aFile width justif x -- aFile
<<wjp         \ aFile width justif precision aNumber -- aFile
```

Writing a buffer (aMemBuffer, aString, ...) into a file is done using #<<

```
aFile "abcdef" <<
```

In order to read from a file, methods used are :

```
atEnd         \ aFile -- b        Returns true if end of file is reached.
>>            \ ( aFile -- c )    Read a character from a file
```

#>> reads a file and returns an integer. The value returned depends on the file mode :

- For File.BINARY and File.TXT, a byte is returned.
- For File.UTF8, the unicode code of the next UTF8 encoded char is returned.

Multiple bytes or chars can be read at the same time :

```
readWith      \ n aMemBuffer aFile -- aMemBuffer
read          \ n aFile -- aMemBuffer

readCharsWith \ n aSB aFile -- aSB : Store characters into a stringBuffer
readChars     \ n aFile -- aSB

readLineWith  \ aBuffer aFile -- aBuffer | null
readLine      \ aFile -- aString | aBuffer | null
```

#readWith and #read are dedicated to read non UTF8 chars. They read n bytes and populate a MemBuffer (#read creates a new MemBuffer).

#readCharsWith and #readChars are dedicated to read UTF8 characters. They populates a StringBuffer.

For all these methods, the number of bytes of characters read can less then than the number asked. The effective number is the size of the object returned.

#readLineWith and #readLine read a file line by line.

#readLine is used to implement a #forEachNext for text files. It loops on each lines into the file. So #forEach: and all higher order functions can be used on a file :

```
"myfile.txt" File new map(#[ words first ]) .
```

This will return an Array of the first word of each line of file "myfile.txt", null if none.

```
"myfile.txt" File new map(#yourself) const: LINES
```

This will create a constant LINES which value is the list of all lines of "myfile.txt".

11.3 Basic input/output

At startup, two files are created and open. They are affected to constants :

```
system.Out          Output defined when Oforth is launched.
System.Err          Error defined when Oforth is launched.
```

Basic output words use System.Out :

```
emit          \ n --       Print the character corresponding to unicode n
type          \ s --       Send string s to System.Out
.             \ x --       Print object x, then a space
.cr           \ x --       Print object x, then perform a carriage return
princr        \ --         Perform a carriage return
```

11.4 Console

If standard streams are not redirected when Oforth is launched, a console object is created and available as a constant :

```
System.Console
```

The console allows to wrap standard input, output and error. Unlike standard streams, the console is a resource, so a task waiting for the console will enter in WAIT state until the console is ready (a key is available,). See the "Concurrent programming" chapter for more information.

At startup, basic words are defined to work with the console are :

```
ekey          \ aConsole -- n     Waits until a key is pressed (including extended).
key           \ aConsole -- n     Waits until a character is pressed.
```

To use extended words, you must import the console package :

```
import: console
```

Currently, cursor handling is not supported; this will be added in a later version.

Constants defined in the console package give a name to the values returned when an extended key is pressed:

```
K-CHAR-MASK K-CTLR-MASK K-ALT-MASK
K-PRIOR K-NEXT K-END K-HOME K-LEFT K-UP K-RIGHT K-DOWN K-INSERT K-DELETE
K-F1 K-F2 K-F3 K-F4 K-F5 K-F6 K-F7 K-F8 K-F9 K-F11 K-F12
```

Methods and functions defined for a console are :

```
ekey?          \ aConsole -- b     Returns true is a character is available
receiveTimeout\ timeout aConsole  --   Waits until a key is pressed or timeout occurs

sendChar       \ n aConsole --     Send a character to the console
sendString     \ s aConsole --     Send a string to the console
flush          \ aConsole --       Removes all pending characters from console input
accept         \ aConsole -- s     Accept a string from the console
fill           \ s aConsole --     Fill console input buffer with string s
```

For instance, if you want to wait for a key during 2 seconds :

```
: wait2s              \ -- x | null
  2000000 System.Console receiveTimeout ;
```

If you want to read an integer (or null is the string is not an integer) :

```
System.Console accept asInteger
```

12 Environment

Oforth is running on top of an OS (at least for now). Some information about the environment are retrieved at startup and available in programs through constants or functions :

12.1 Environment constants

Here are the constants set at startup :

```
System.VERSION       \ -- s       Returns Oforth version.
System.ISWIN         \ -- b
System.ISLINUX       \ -- b
System.ISMAC         \ -- b

System.CORES         \ -- n       Number of cores detected.
System.MAXWORKERS    \ -- n       Max number of workers that the VM will create.

System.Args          \ -- aArray  Array of command line arguments (only with -).
System.Paths         \ -- aArray  Array of paths retrieved from OFORTH_PATH env var.
System.Console       \ -- aConsole Oforth console (null if no console).

System.AssertMode    \ -- b       True if --a option
System.Optimize      \ -- b       True if not --C option.
System.TestMode      \ -- b       True if --t option
System.Unsecure      \ -- b       True if --U option.

System.Out           \ -- aFile   Standard output file
System.Err           \ -- aFile   Standard error file
```

12.2 Functions

```
System.task          \ -- aTask   Returns current task running.
```

Function System.tick can be used to calculate elapsed time between two ticks.

```
System.tick          \ -- u       Return a tick in microsecond
```

To retrieve number of microseconds since 01/01/1970 :

```
System.time          ( -- u )
System.localTime     ( -- dst min u )
```

System.localTime returns :

- dst : boolean that say if daylight saving time.
- min : number of minutes between utc time and local time.
- u : same as System.time (number of microseconds since 01/01/1970).

13 Concurrent programming

Oforth is designed for concurrent programming. It implements a task/channel model : a task is a piece of code that can run concurrently with other tasks. Each task is isolated and can communicate with others only using channels.

A channel is a structure that allows tasks to send objects and for other tasks to receive them.

13.1 Immutability and task isolation

Apart from objects that are sent into channels, memory used by a task is isolated : other tasks can't see mutable objects created by a task. Only immutable objects are visible. Two tasks can't update the same object at the same time, so a task is assured that its mutable objects are only visible by itself.

This is done by design :

- Oforth has no global variables that could hold a mutable object visible by tasks.
- Oforth has no class attributes.
- tvar values are by tasks.
- Constant values can't be mutable (an exception is raised if you try of create a constant with a mutable value).
- Channels only accept immutable objects.
- An immutable attribute value can't be a mutable object.
- Attributes value of an immutable object (ie an object with only immutable attributes) can't change after the object is initialized.

All this is checked at runtime and an exception is raised if a problem occurs.

The first consequence of this model is that objects are never copied when sent to a channel. As this object is immutable, there is no way for a task receiving it to update it.

The second consequence is that there is no mechanism such as mutexes, semaphores, ... Those mechanisms are not necessary as there is no situation where a mutable object can be updated by another task than the one that created it.

13.2 Threads and workers

In Oforth, threads are not exposed to the programmer : they are handled automatically by the virtual machine.

The programmer creates tasks and the VM chooses (or creates if necessary) a thread to run this task. If a task is paused because it waits for an event or a resource (a channel, a socket, a console, ...), this does not block the thread the task was running on : the thread is automatically affected to run another resumable task.

Those threads are named workers. At startup, one worker is launched, running the interpreter. If tasks are resumable and no worker is available, the VM creates a new worker. By default, the maximum number of workers to create is the number of cores available on the machine. This can be changed used a command line option :

```
--Wn          \ Defines n as the maximum number of worker created by the VM.
```

13.3 Tasks

Tasks are a piece of code that will run in parallel with other tasks. If the system has more than one core/processor, tasks will truly run in parallel. Otherwise, each task will use a part of the CPU of the processor.

Unlike OS threads, tasks are very light objects. Creating a task is not heavier than creating a (big) object.

A task has its own data stack. The data stack is not shared by tasks.

Creating a task is done using #& word. This word requires to import the **parallel** package :

```
&       \ r --       : Creates and launch a new task that will run r in parallel.
```

Launching a task does not mean that this task will run immediately. The task is tagged as resumable and will run when a worker is available. If a task stay some time in resumable state, the VM can decide (if possible) to create another worker.

For instance :

```
import: parallel
: helloWorld   "Hello, World\n" . ;

#helloWorld &                    \ Launches function #helloWorld in parallel
#[ 10 #helloWorld times ] &      \ Launches block in parallel (running 10 hw).
10 #[ #helloWorld & ] times   \ Launches 10 tasks each running #helloWorld in parallel
```

Sending parameters to a task is done by using a closure : a closure will keep values that will be used by the task. Of course, those values can't be mutable (if so an exception is raised). You can create a closure with mutable values, but, if so, you can't run it in parallel.

```
: hello             \ s --
    "Hello," . .cr ;


: test( s -- )      \ Launches a new task
    #[ s hello ] & ;

"Franck" test                           \ ok
StringBuffer new "Franck" << test       \ ko : an exception is raised
```

Task can be created explicitly instead of using #&, by providing a runnable :

```
newSize       \ r n Task -- aTask   Creates a new task with n as data stack size
new           \ r Task -- aTask     Creates a new task with 200 as data stack size
```

Methods defined for tasks are :

```
runnable      \ aTask -- r        Returns task's runnable
schedule      \ aTask             Shedule a task. The tasks will run on a worker.
System.suspend\ -                 Suspend current task until it is resumed.
resume        \ aTask             Resume a suspended task.
```

With tasks, a new method is added to map collections with calculations done in separate tasks running in parallel :

```
mapParallel   \ r coll -- aArray
```

#mapParallel will map r on each element of coll. Each application of r on an element is done into a dedicated task running in parallel. #mapParallel returns when all calculations are done :

```
1000 seq mapParallel(#sqrt) sum   \ Each sqrt is done in a separate task
```

13.4 Channels

A channel is a way to communicate between tasks. It is a structure dedicated for sending and receiving objects. Multiple tasks can send to the same channel and multiple tasks can receive from the same channel. Only immutable objects can be sent into a channel.

```
newSize       \ n Channel -- aChannel  : creates a new channel with size n
```

```
new            \ Channel -- aChannel    : creates a new channel with defaut size (200)
```

After a channel is created, the channel is open and objects can be sent into or receive from a channel :

```
send           \ x aChannel -- b    : Send x into aChannel
```

#send sends an immutable object into a channel. If the channel is full, the task will wait until there is room into the channel. If the send is successful, return is true. If the channel is closed, return is false.

```
sendTimeout    \ x n aChannel -- b : Send x into the channel with n as timeout.
```

Same as #send, but with a timeout (in microseconds) as parameter. #sendTimeout with return false if the channel is closed or if the timeout is reached. Sending null as timeout value means "no timeout", so this will be the same behavior as #send.

```
receive        \ aChannel -- x | null : Receives an object from a channel.
```

#receive retrieves an object from a channel. If the channel is empty, the task waits until an object is present into the channel. null is returned if the channel is closed AND if the channel is empty.

```
receiveTimeout \ aChannel -- x | null : Receives object from a channel with timeout.
```

Same as #receive but, if the channel is empty, it waits until an object is available into the channel or if the timeout is expired. If so, it returns null.

A task can receive objects from a closed channel while it is not empty (but a task can't send an object into a closed channel).

Channels themselves represent the transport of objects between tasks, not the objects themselves. So a channel is an immutable object and can be the value of a constant or sent as parameter to a task. For instance :

```
import: parallel

Channel new const: MyMailBox


: job                \ --
   1 2 + 10000 sleep MyMailBox send drop ;


#job &

 MYMailBox receive .
```

It is common to send a channel as parameter to another task. This allows to specify on which channel this task will send or receive objects. As channels are immutable objects, they can be values into a closure.

Here is a ping pong between two tasks where the channels used are created prior to running the tasks :

```
import: parallel

: pong(n  ch1  ch2 -- )
   | i | n loop: i [ ch1 receive sqrt ch2 send drop ] ;

: ping (n -- )
| ch1 ch2 i |
   Channel new ->ch1
   Channel new ->ch2
   #[ n ch1 ch2 pong ] &
   n loop: i [ i ch1 send drop ch2 receive . ] ;

100 ping
```

The #ping function creates 2 channels, then launches a new task running #pong in parallel. It uses a closure to send 3 parameters to #pong function : n, ch1 and ch2. Then it sends integers from 1 to n into channel ch1, waits for the answer from ch2 and print the object returned.

The #pong function loops n times : it waits until an object is available on channel ch1 and, when received, calculates #sqrt and sends the result to channel ch2.

A channel is the only object that allows to synchronize tasks. A channel can be used for various purposes :

- Manage a log file where multiple tasks can write in parallel (see logger package).
- Manage responses to events (see emitter package).
- Manage servers (see tcp package)
- ...

13.5 Resources

A task can wait for a resource to be available. If so, the task is stopped until the resource is available and the worker can run other tasks. A task waiting for a resource consumes no CPU : the task is blocked (but not the thread).

Channels are resources, but they are not the only resources defined. Other resources are :

- Waiting for a duration (#sleep)
- The console input / output
- Sockets.

When a task is asking for a resource and this resource is not available (channel empty or full, no input, no data on the socket, ...), the task enters into a WAIT state and is no more resumable until the resource is available. When the resource is available, the VM detects the event and set the task

back to a resumable state. The task will run when a thread is ready to run it.

You can ask the current task to sleep using #sleep

```
sleep          \ n --       The current task will sleep for n milliseconds.
```

The task will not resume before n milliseconds.

The console is a resource, so a task waiting for console input does not block any thread. You can ask the current task to wait for console input :

```
System.ekey    \ -- n       : Waits for an extended key pressed and returns its code
System.key           \ -- n       : Waits for a key pressed and returns its code.
accept         \ aConsole -- aString : Waits for a string and return it.
```

Sockets are also resources. Reading or writing on a socket will block the task (but not the thread) if the socket is not ready for the operation. See the socket package for more information.

13.6 Tasks reporting

#.t word prints the status of all task running into the system and, if tasks are waiting, what resource they are waiting for. If you try #.t at startup, you will probably see this kind of output :

```
>.t
TASK                                STAT WORKR WAIT
#interpreter                        RUN      1
```

This means that the system is running one task, the interpreter and this task is running. This is normal as the interpreter is currently running #.t function.

Let's try to identify on what resource the interpreter task is waiting for when we don't enter a command :

```
>import: parallel
>#.t &
ok
TASK                                STAT WORKR WAIT
#.t                                 RUN      1
#interpreter                        WAIT       consoleReceiveTimeout
```

Instead of running #.t, we run it in parallel into another task. Now we see two tasks. There is the

task running #.t that we have just launched. And there is the task running the interpreter that is in WAIT state. To the right we see on which resource the task is waiting for. Here, the interpreter is waiting for input from the console : it waits for a key pressed on the keyboard. While no key is pressed, the interpreter stay in WAIT state and does not consume CPU.

13.7 Summary

Oforth concurrent programming model characteristics are :

- The programmer does not use threads directly; only tasks are used and they are light compared to threads.
- The VM manages the threads and distributes tasks among threads.
- Only immutable objects can be shared between tasks.
- There is no need to synchronize data access, as two tasks can't update the same data at the same time.
- There is only one object to synchronize tasks : the channel.
- The channel itself is an immutable object and can be shared among tasks.
- Resources are channels, sockets, console, ...
- Tasks waiting for resources are in WAIT state and does not block the workers.

14 Memory

Memory access is an area with the most differences with Standard Forth.

Oforth has no pointer to addresses. Well, that's not true, as pointers are everywhere. But they are not exposed. Pointers are replaced by objects. You don't have direct access to memory other than accessing to objects attributes (which are objects, too).

You create objects and the garbage collector is responsible to delete them when they are no more used.

14.1 Memory areas

Memory addressed by the system is divided into various areas :

- Dictionary : it is the area where created words are stored.
- Code space : it is the area where definitions are compiled into native code.
- Data stack : it is the area that holds parameters (one data stack by task).
- Return stack : a task return stack is handled by the worker's execution stack.
- Heap : it is the area where objects are created. There is one heap by worker. When a task running on a worker wants to create a new object (with #new), it asks its worker to create it.

14.2 Allocating/freeing memory

Memory is allocated when new objects are created :

```
new           \ ( aClass -- aObject )
Creates a new object of class aClass on the heap and push it on the stack.


allot         \ ( aClass -- aObject )
Creates a new object of class aClass in the dictionary and push it on the stack
```

When the object is allocated, these methods call #initialize with this object as receiver. Parameters needed to initialize attributes can be retrieved on the stack.

There is no other way to allocate memory in the dictionary but to use #allot. Dictionary holds

objects too.

Objects allocated can't be manually freed : for objects created on the heap, the garbage collector will free them automatically. For objects created in the dictionary, they will live forever. It is possible to forget a word into the dictionary using #forget (see Words).

The GC is an incremental Mark and Sweep GC : tasks are allowed to run while the GC is running.

If interested by the GC's internals or GC parameters, you can find information here :

http://www.oforth.com/memory.html

15 Exceptions

Exceptions are objects that can be thrown when an exceptional event occurs.

Some built-in functions generate exceptions and a program can generate its own exceptions.

When an exception is thrown, the current execution stops and the program restart to a point where the exception is catched.

15.1 Catching exceptions

The text interpreter catches all exceptions; if an exception is not catched by the program, it will be catched by the text interpreter :

```
: test1 1.2 0 / ;
: test   test1 ;
test
```

User defined exceptions, if not catched, will be catched by the interpreter too :

```
: test1 "This is my exception" Exception throw ;
: test   test1 ;
test
```

A structure try/when allows to catch an exception and decide what to do with it. A try/when structure needs a local variable to be declared. When the exception is catched, the local value is the catched exception into the when block :

```
try: e [ instructions1 ] when: [ instructions2 ]
```

If instructions1 throw an exception, the program stops and execute intructions2 into the when block.

If instructions1 don't throw an exception, instructions2 are not executed and the program continue after the when block.

There are three ways to handle a catched exception into the when block :

1) You can do some work (log, print, clean, ...) and throw the exception again for another try/when block (or the interpreter) :

```
: test1       1.2 0 / ;
```

```
: test | e | try: e [ test1 ] when: [ "I catched you, " . e . e throw ]
```

2) You can do some work and continue

```
: test1       "An exception" Exception throw ;
: test
| i e |
 10 loop: i [
    try: e [ test1 ] when: [ "I catched you, but I don't care" .cr ]
         ]
;
```

3) If you want to catch only some kind of exception you can check the exception type (i#sA or #isKindOf) to decide what to do (handle or throw again).

15.2 Exception class

Exception class is the base class exceptions.

Methods implemented are :

```
throw         \ s Exception --     : Creates and throw an exception with message s.

throw         \ aException --      : Throw an exception
message       \ aException -- s    : Returns the exception's message.
log           \ aException --      : Log an exception on standard error.
```

All exceptions inherit from this class.

15.3 Predefined exceptions

Some built-in functions or methods already throw exceptions :

```
Exception     \ A basic exception with a message.
ExRuntime     \ Exception that occurs at runtime with a message and an object
ExBadType     \ Exception thrown after a type check by
ExIO          \ Exception during I/O (file, socket, ...).
```

```
ExMath          \ Math exception (divide by zero, bad range, ...).
```

16 Packages

Two words can't have the same name : if you try to do this, you will raise an exception. This rule is strict and, if you load many features, name conflicts may appear.

In order to handle this, Oforth implements packages. A package defines :

- A list of files to load to implement a functionality.
- A namespace for the words declared into those files : 1 package = 1 namespace

Two words can have the same name if they are declared into two different packages. A word name can be prefixed with its package to avoid ambiguity when necessary :

```
12 dup .s
13 oforth:dup .s
```

oforth is the package where all built-in words are declared.

At any time, the system has a current package, where all new words are created. When the interpreter starts, the current package is **oforth**.

Package's themselves are words declared into the oforth package :

```
import: date
date .s
oforth:date .s
```

16.1 Loading a package

A package is loaded using #import: or #use:

```
import:             \ "name" --         : Import the package named "name"
use:                \ "name" --         : Use the package named "name".


import: date
use: tcp
```

#use: loads into the system the package which name is into the input buffer.

- If this package is already loaded, #uses: does nothing and returns.
- It creates a new package into the **oforth** namespace.
- It changes the current package to this package

- It loads all package's files into this namespace.
- It restore the current package to the previous value.

With #uses: , words created must be prefixed with the package name. Otherwise, they are not found :

```
use: date
Date .s              \ Not found, words must be prefixed
date:Date .s         \ ok
```

It is possible to create aliases to those words :

```
#date.Date alias: Date
```

A package can also be loaded using #import:

```
import: package_name
```

This function does the same thing than #use: but the package loaded (package_name) is declared as an imported package of the current package. Into this current package, words imported can be used without prefixing them :

```
import: date
Date .s
```

Here, package **date** is imported into the current package (probably **oforth** package, unless imported from another package). So date package is declared as an imported package of oforth and its words can be accessed without prefixing them (you can prefix them if you want or to avoid ambiguity).

```
Date now .s
```

A package has a list of imported packages. When this package is the current package (ie when it is loaded into the system), each package loaded using "import:" is added to this list. All words declared into a imported package don't have to be prefixed with the package's name.

16.2 Search for packages

To load a package, the package file must be found. This file includes a description of the package (as a Json object). This file is searched :

- Into directories listed into the OFORTH_PATH variable value.
- Into a "packs" directory into these directories.

16.3 Search order for words

Search for words occurs when # is used (the name is read from the input buffer) or using #find word (the name is into a string provided as parameter).

If the name is qualified (ie prefixed by its package), the search is done only into this package.

If the name is not qualified, the search is extended :

- To the current package.
- If not found, to imported packages of current package.
- If not found, to the oforth package.
- If not found, to imported packages of oforth package.
-

Consequences :

- Names created into a package can be used into this package without prefixing them.
- Importing a package just adds it to the list of the imported packages of the current package. If oforth package imports a tcp package and if tcp package imports logger package, words defined into logger must be prefixed into oforth (unless oforth itself imports logger package).
- There is no conflict error when a non qualified word is used and has a name declared more than once. The first word found will be used.

16.4 Creating a package

To create a package, you must :

- Create a file with a Json object describing this package.
- Put this file into one directory included into the OFORTH_PATH variable.
- Create a new directory into this directory with the same name as the one used into the Json.
- Put the package files sources into this directory.
-

For instance, to create a package named "parallel", you put this Json into a file named "parallel.pkg" :

```
{
   $signature : {
      $name         : "parallel",
      $version      : "V1.0.0",
      $state        : $Stable,
      $directory    : "parallel",
```

```
        $author       : "Franck Bensusan",
        $contributors : [ ],
        $url          : "www.oforth.com",
        $license      : "BSD"
        } ,
    $files : [ "Resource.of", "Channel.of", "Task.of" ]
}
```

This Json lists 3 files than must be found into the $directory directory (here parallel). So :

- The file parallel.pkg containing this Json must be into a directory listed into OFORTH_PATH variable.
- This directory must have a sub-directory named **parallel**
- This sub directory must have 3 files : Resource.of, Channel.of and Task.of

When this package is imported using :

```
import: parallel
```

the 3 files are loaded in the order they appear into the $files list.

17 Structures and FFI

Oforth allows to define structures like C structures and call functions into dynamic libraries.

Currently, structures are not fully implemented, but allows to handle majority of FFI calls.

17.1 Structures

A structure is created using #struct word. Structure name, fields types and names follows. Types currently supported are ptr (an address) and int (an integer). For instance, a MYSQL_FIELD declaration is :

```
struct MYSQL_FIELD
  ptr    name                \ Name of column
  ptr    org_name            \ Original column name, if an alias
  ptr    table               \ Table of column if column was a field
  ptr    org_table           \ Org table name, if table was an alias
  ptr    db                  \ Database for table
  ptr    catalog             \ Catalog for table
  ptr    def                 \ Default value (set by mysql_list_fields)
  int    length        \ Width of column (create length)
  int    max_length          \ Max width for selected set
  int    name_length
  int    org_name_length
  int    table_length
  int    org_table_length
  int    db_length
  int    catalog_length
  int    def_length
  int    flags               \ Div flags
  int    decimals            \ Number of decimals in field
  int    charsetnr           \ Character set
  int    type               \ Type of field. See mysql_com.h for types
;
```

This creates a new word of type Struct. Struct objects respond to :

```
size                \ aStruct -- n      Returns structure size in bytes.
```

You don't create structures. The choice was to use integers (as pointers) and/or MemBuffers and structure offsets to retrieve value or set values. Two functions are provided to do this :

```
at: STRUCT->field    \ aMemBuffer | aInteger -- x
Returns the value at offset STRUCT->field

put: STRUCT->field   \ x aMemBuffer --
Set value at offset STRUCT->field with x
```

For instance, to retrieve information from an MYSQL_FIELD :

```
SQLManager_MySQL method: _fetch_field
| fieldptr |
    _mysql_fetch_field ->fieldptr
    fieldptr at: MYSQL_FIELD->type
    fieldptr at: MYSQL_FIELD->max_length
    fieldptr at: MYSQL_FIELD->name tostr asSymbol
    SQLField new
;
```

17.2 Dynamic Libraries

Dynamic libraries are words. They are created and loaded using #new:

```
new:          \ str typeparam OSLib "name" --
```

This creates and loads a new dynamic library with :

- str is the file name of the library to load
- typeparam is WPARAM or CPARAM according to how parameters are handled.
- "name" is the Oforth word name for this library.

For instance :

```
"libmysql.dll" WPARAM OSLib new: LIBMYSQL
```

or

```
"libmysqlclient_r.so.18" CPARAM OSLib new: LIBMYSQL
```

Here a new dynamic library is created with LIBMYSQL as name.

17.3 Dynamic functions

Once a library is created, dynamic functions can be created. They are also words are they are called like functions or methods. A dynamic function is created like this :

```
new:          \ aOSLib nbParam returnClass procname OSProc "name" --
```

This creates a new dynamic function :

- OSLib is the libray
- nbParam is the number of parameters to take on the data stack to call the function.
- returnClass is the class of the object returned. null means that nothing is returned.
- procname is a string corresponding to the name of the function into the library.
- "name" is the name of the OSProc word.

For instance, some declarations of mysql dynamic functions :

```
LIBMYSQL 1 Integer "mysql_init"            OSProc new: _mysql_init
LIBMYSQL 8 Integer "mysql_real_connect"    OSProc new: _mysql_connect
LIBMYSQL 1 null    "mysql_close"           OSProc new: _mysql_close
LIBMYSQL 2 Integer "mysql_query"           OSProc new: _mysql_query
LIBMYSQL 1 Integer "mysql_num_fields"      OSProc new: _mysql_num_fields
LIBMYSQL 1 String  "mysql_error"           OSProc new: _mysql_error
LIBMYSQL 1 Integer "mysql_fetch_field"     OSProc new: _mysql_fetch_field
LIBMYSQL 1 Integer "mysql_fetch_row"       OSProc new: _mysql_fetch_row
LIBMYSQL 1 Integer "mysql_store_result"    OSProc new: _mysql_store_result
LIBMYSQL 1 Integer "mysql_use_result"      OSProc new: _mysql_use_result
LIBMYSQL 1 null    "mysql_free_result"     OSProc new: _mysql_free_result
LIBMYSQL 1 Integer "mysql_affected_rows"   OSProc new: _mysql_affected_rows
```

Once an OSProc is created, it is called as any other runnable (no need to use #perform). For instance :

```
SQLManager_MySQL method: _close
 _sqlConnectionPtr _mysql_close ;


aSQLResult _resPtr _mysql_fetch_row
```

18 Ans Forth / Oforth cross reference

This chapter lists most Ans Forth words and Oforth counterpart, with commentaries if necessary.

Creating words and compilation

:	:	
	method:	Creates a method implementation
	virtual:	Creates a virtual method implementation
	classMethod:	Creates a class method implementation
	classVirtual:	Creates a class virtual implementation
;	;	
immediate	immediate	
{ } {: :}	( )	Oforth uses ( ) to declare local parameters { : , and } are used for Json objects syntax {: and :} are ... not defined
|	|	Local variables are declared between two |
[	<%	[and] are used for arrays
]	%>	[and] are used for arrays
[: :noname	#[	
;]	]	
variable 2variable fvariable		Not defined. No global variable is permitted in Oforth.
user	tvar:	Value of a tvar is by task.
value 2value	tvar:	Value of a tvar is by task.
to	to ->	to is used for setting tvars, -> is used for setting locals
constant 2constant fconstant	const:	
alias	alias:	
create does> >body		Not defined. Replaced by classes definition.
' [']	#	# returns an object on the stack, not an execution token.
find find-name	Word find	

>name	code	
name>string	name	
execute	perform	Oforth has no xt, so execute is not defined
compile,	compile	
evaluate	evaluate	
quit abort		Not defined
abort"	abort	
throw	throw	throw need an Exception object
catch		Not defined
try ... endtry	try: [...] when: [...]	
defer	defer:	
defer@ action-of	action	
is defer!	act	is is to check if a class is of a property.
postpone	compile perform	
literal 2literal fliteral sliteral	literal	literal is not immediate.
state	STATE	
source >IN		Not implemented
parse parse-name	parse-name	
refill	fill	Fill console input buffer with a string value.
include	load	
required	import: use:	
see	see	
bye	bye	
forth	oforth	oforth is the default package. Unlike forth, its name just pushes it on the stack.
environment ?		Not defined

Stack manipulation :

dup fdup	dup	
drop fdrop	drop	
over fover	over	
swap fswap	swap	
rot frot	rot	
-rot	-rot	

nip fnip	nip	
tuck	tuck	
2dup	2dup	
2drop	2drop	
pick fpick	pick	
depth fdepth	.depth	
.s	.s	
roll		Not defined
2nip		Not defined
2over		Not defined
2swap		Not defined
2rot		Not defined
?dup		Not defined

Arithmetic :

+ f+ d+ m+	+	
- f- d-	-	
* f* m*	*	
/ f/	/	
mod	mod	
/mod um/mod fm/mod	/mod	
negate fnegate dnegate	neg	
abs dabs fabs	abs	
sqrt fsqrt	sqrt	
max dmax fmax	max	
min dmin fmin	min	
1+	1+	
1-	1-	
2* d2* f2*	2*	
2/ d2/ f2/	2/	
s>d d>s		Not defined. Oforth integers have arbitrary precision
d>f s>f	asFloat	
f>d f>s	asInteger	
f**	powf	
fexp	exp	
fln	ln	
flog	log	
1/f	inv	
fsin	sin	

fcos	cos	
ftan	tan	
fasin	asin	
facos	acos	
fatan	atan	
fsinh	sinh	Need import: math
fcosh	cosh	Need import: math
ftanh	tanh	Need import: math
fasinh	asinh	Need import: math
facosh	acosh	Need import: math
fatanh	atanh	Need import: math
f~	==	

Conditions :

true	true	
false	false	
and	and bitAnd	
or	or bitOr	
xor	xor bitXor	
invert not	not	not returns true if false and false otherwise
lshift	bitLeft	Be careful, in Oforth stack effect is inverted (method of integer)
rshift	bitRight	Be careful, in Oforth stack effect is inverted (method of integer)
= d=	=	
f~ compare	==	Test objects values
< u< d< du< f<	<	
<= f<=	<=	
<>	<>	Carefull, Oforth compare by value.
> u> f>	>	
>= f>=	>=	
0= 0<> 0> 0< 0<> d0< d0= f0< f0=		Not defined

Control flow :

\ ( ) --	\ --	In Oforth, () are used to declare parameters
if	ifTrue: [ifFalse: [ifZero: [ifNull: [ifNotNull:[if=: [if	IfTrue: [...] if ... else ... then is also defined in Oforth.
then repeat until	] then	
else	else: [else	ifTrue: [...] else: [...]
begin again	begin again	
begin ... while ... repeat	while (...) [...]	
begin ... until	doWhile: [..]	
ahead	ahead	
cs-pick cs- roll	CS> >CS	
case of endcase		Not defined
do ?do +do u+do -do loop +loop	loop: i [...] -loop: i [...] for: i [...]	
I J		Not defined. Oforth loops require a local declaration.
recurse		Not defined, use function name to recurse
leave	break	
	continue	
exit	return	
>R R@ >R rdrop 2>R 2R> 2R@ n>r 2RDROP		Not defined. Locals handles values on the return stack.
[IF]	IFTRUE: [IFFALSE: [	
[ENDIF]	]	
[DEFINED]	DEFINED:	

I/O :

. u. ." .id d. ud. f.	.	. applies to all objects

emit	emit	
ekey	ekey	ekey is done on a console object
key	key	key is done on a console object
base hex decimal	0x 0b	base hex decimal are not defined
.r		
<# <<# # #s #> #>> hold	<< <<w <<wj <<wjp	
bl	BL	
space		Not defined.
cr	printcr	
accept	accept	accept is done on a console object
ms	sleep	
time&date	System.local Time System.time	

Characters and Strings :

char [char]	'	No need to have a space after '
s"	"	No need to have a space after "
c, c@	at put	
type	type	
compare	==	
search	indexOfAll	
fill	<<cn	
-trailing	strip	
s>number? s>unumber? >number	asInteger	
>float	asFloat	
count		Not defined
substitute	replaceAll	

Memory :

@ ! , f@ f! +! c@ c! 2@ 2! sf@ sf! df@ df!		Not defined : you can't make direct access to memory
	@ :=	Access to an object attributes

	at: put:	Access to a structure fields
Chars char+		Not defined
cells floats	cells	
cell+ float+	cell+	
allot	allot	You can only create objects in dictionary
allocate	new	You can only create objects on the heap
free		No direct free is possible. Memory is handled by GC.
resize		Not defined : if resize is possible (ArrayBuffer, ...) resize automatically when needed
align aligned faligned sfalign dfalign		Not defined. In Oforth all objects are aligned.
here c, f, , 2,		Not defined. Either new or allot.
ADDRESS- UNIT-BITS	CELLSIZE	
move erase cmove cmove>		Not defined
buffer:		Not defined. Can't define a mutable word. Use : MemBuffer newSize to create a buffer on the heap
begin- structure	struct	
end- structure	;	
field:		Not defined : structure fields are declared without field:

Files :

r/o	File.READ	
w/o	File.WRITE	
r/w	File.APPEND	
bin	File.BINARY File.TEXT File.UTF8	
Create- file	newMode new	
open-file	open	
close-file	close	
read-file	>> readWith read readCharsWith readChars	

read-line	readLineWith readLine	
write-line	<<	
write-file	add addChar << <<w <<wj <<wjp	
flush-file	flush	
file-status	exists	
file-position	position	
reposition-file	setPosition	
file-size	size	
stdin	Console	
stdout	System.Out	
stderr	System.Err	

Vocabularies and word lists :

Word lists words are not used in Oforth. Oforth implements packages instead and they are too different from lists or vocabularies to have a cross reference.