The fOOrth User's Guide and Reference

An RPN, object oriented language that is not FORTH.

by Peter Camilleri

Last Update: July 25, 2015

Covering fOOrth version 0.3.0

Status: Preliminary

Table of Contents

The MIT License (MIT).

Introduction

Thank you for taking a moment to peruse the fOOrth user's guide and reference. The following pages, chapters, and sections deal with fOOrth, an experiment in FORTH inspired language design and compiler implementation in Ruby. In particular, a special focus on object oriented design and meta-programming was applied to the language and its implementation.

It must be stressed that as an experiment, it is likely the fOOrth is not especially suited to any particular purpose, aside from research. Then again, perhaps some use beyond academic interest will be found.

On a related matter, as an esoteric research language, neither fOOrth nor this User's Guide is a good programming introduction for a beginner or early programmer. On the contrary, the “raw frontier” nature of this work makes it far more suitable to those well versed in a three or more languages, or at the very least, having an in depth knowledge of Ruby (that's Ruby, not Ruby on Rails, see Ruby below) or Smalltalk.

Again, thank you for your interest; Any comments, suggestions, fixes, improvements, or criticisms are most welcomed.

About the name fOOrth

The name of programming language fOOrth is an example of a malamanteau[1]. That is a portmanteau of a malapropism.

The portmanteau portion of this is the mash-up of FORTH[2] and the “OO” of object oriented programming[3] systems. It is short, easy to pronounce, slightly witty, and a unique opportunity to describe an actual word as being a malamanteau.

The malapropism involved is quite simply that fOOrth is not FORTH. While the acronym FNF, for **f**OOrth is **N**ot **F**ORTH, is also short, it is not at all easy to pronounce as fOOrth.

How fOOrth came to be

I have had a fascination with stack based, postfix notation programming environments going back over 30 years. This started with those awesome calculators by Hewlett Packard[4]. I could never afford to buy one, but I was always intrigued by their elegant, expressive power compared with more conventional calculators.

Later, as part of a college course I was tasked with creating a programming language interpreter. I must admit I was struggling with this task. Then one Sunday afternoon, while visiting my Uncle Sal, I began working away at his Radio Shack TRS-80 Computer in Basic. In the course of a two hour programming session, I had created a tiny stack based calculator. Inspired by the simplicity of this simple interpreter, I was able to design a fuller version that ran on the University's PDP-10 mainframe in APL. Yes it ran fairly slowly. The odd thing was that

1 Malamanteau, see XKCD 739 at http://xkcd.com/739/, http://en.wikipedia.org/wiki/Xkcd#Recurring_items
2 See http://en.wikipedia.org/wiki/Forth_(programming_language), http://en.wikipedia.org/wiki/Charles_H._Moore
3 See http://en.wikipedia.org/wiki/Object-oriented_programming
4 See http://en.wikipedia.org/wiki/Hewlett-Packard, http://en.wikipedia.org/wiki/HP_calculators, and http://www.hpmuseum.org/

compared to the other student's efforts, it ran amazingly *fast*! The simplicity of the syntax meant that little time was wasted parsing and analyzing the source text.

Some years later, I was involved in a major project developed in FORTH during which I came to admire and appreciate the expressive power of the language. While the project was not successful in the end, this was largely a consequence of the nearly impossible goals we had set for ourselves and not a reflection of FORTH. Well not much of a reflection on FORTH as we'll see next.

FORTH today is about as dead a language as you are going to find. Like other extinctions throughout history, this can be traced by its utter failure to adapt to changing conditions. When FORTH was born, most computers were very primitive. They had very little processing power, microscopic memory resources and mass storage, and lacked any form of operating system, file storage, or peripherals; user interfaces were upper case only ASCII subsets. In this environment FORTH was an excellent choice. Systems were tiny and FORTH fit those systems well, and its supporters liked things that way. As computers improved, FORTH stood still. File systems, floating point math, memory management, improved user interfaces all came to everyday computers, but not to FORTH[5].

FORTH was still efficient and fast, but computers continued to get more and more memory, processing power, and advanced I/O. FORTH was static, stationary, and dying. To this day, FORTH is *still* largely an upper case only language. Just recently. I remember watching in disbelief a video in which a spokesperson for the company Green Arrays, Inc[6] stated that an enhancement to their FORTH oriented processor would be to *limit* the addressable memory to a mere 64 words. How out of touch with reality do you have to be to think that such limited memory is an *asset*?

So if FORTH is so stone age, primitive and extinct, why do this fOOrth thing? It goes back to the heart of fOOrth and FNF. Remember, fOOrth is *not* FORTH. In spite of its problems, the expressive elegance of FORTH cannot be denied. The fOOrth system is an attempt to project where FORTH might have gone had it supporters been more progressive. The fOOrth system is dedicated to destroying limitations, not exalting them.

Given all what has been written, there is still one inescapable fact. FORTH did possess a simplicity and clarity that made it attractive. The fOOrth language needs to retain this essential simplicity as much as possible while avoiding the corner-cutting that made FORTH miserable for many tasks. The fOOrth language needs to retain this sense of small, understated elegance.

Finally, fOOrth is inspired by the development of Object Oriented[7] and Message Passing[8] paradigms[9] pioneered in the Smalltalk[10] programming language. While Smalltalk, is a fairly obscure language today, there can be no doubt of its tremendous impact on modern programming language thought and design.

In the following sections, the underlying principles of fOOrth will be examined to provide a basis for a detailed look at the architecture, features, and facilities of the language system.

---

5 Yes all of these things were added, but always as optional, poorly supported extensions. They were never really part of the core language with the full support they needed.
6 See Green Arrays, Inc. at http://www.greenarraychips.com/
7 See http://en.wikipedia.org/wiki/Object-oriented_programming
8 See http://en.wikipedia.org/wiki/Message_passing
9 See http://en.wikipedia.org/wiki/Paradigm
10 See http://en.wikipedia.org/wiki/Smalltalk

Goals and Principles

To move forward, fOOrth has adopted a few basic goals and principles. These are:

1) A Simple, Easy-to-Understand syntax that is none the less, expressive and compact:
    - Source code in fOOrth is free-form with no line oriented limitations or rules.
    - Support is given to the easy representation of common literal data such as strings, integers, floating point numbers, rational numbers, and complex numbers.
2) Safe Data and Data Structures:
    - Simple, reliable arithmetic. In fOOrth, integer operations never overflow. Rational values can be represented exactly, complex numbers are supported, and conversions between numeric types are simple.
    - Strings grow as needed without the need to allocate space or worry about overflow.
    - Data containers such as arrays and hashes grow as needed. Out of range subscripts cannot access undefined memory regions.
3) Message Passing:
    - In fOOrth all actions take the form of messages sent to a receiver.
    - The routing of messages is specified by the exact type of the message.
    - Message receivers include data items on the stack as well as the virtual machine object associated with the current thread of execution.
    - Messages for which no routing specification can be found, generate an error at compile time.
4) Object Oriented Design:
    - Support class based inheritance, with a non-cyclic (single inheritance) tree derived from a common base Object class.
    - Support late binding and polymorphism through message interface compatibility or “duck” typing.
5) Meta programming:
    - Support extensible language constructs by making the compiler an accessible part of the system.
6) Reliability:
    - As much as is possible, invalid operations should generate errors rather than erroneous results
    - Errors should be detected as soon as possible.
7) Building on the host language:
    - The fOOrth language is built upon the Ruby language. To leverage this, fOOrth has the ability to build proxy connections to Ruby classes and facilities.

About Ruby

The fOOrth language is implemented in Ruby. The reasons for this are very simple:

- Ruby is a powerful, expressive language with introspection and meta-programming tools that make it ideal for creating new programming languages.
- Programming in Ruby is pure joy and I'd rather not be miserable. In fact, one of the stated goals of the language was to maximize programmer joy. Success achieved!

Now I need to make a clarification here. I speak of Ruby, not its popular offspring Ruby on Rails. In the minds of many theses are the same thing; they are not[11].

Ruby is a flexible, powerful language akin to an artist's studio with a wide choice of media, pigments, tools, and techniques to create his masterpieces. Rails is a web server framework, more like a government sponsored art program with strict guidelines, and helpers to “guide” the hand of the artist to produce any work of art so long as it is a portrait of Elvis on black velvet. As you might gather, I really really like Ruby. Ruby on Rails, not so much.

Special Notes of Thanks

Firstly: This project, as well as this User Guide owe a huge debt of gratitude to the Wikipedia[12] project. Throughout this guide, entries from the free encyclopedia are used to illustrate and amplify many crucial concepts. Wikipedia is supported by donations and I urge all to add their support. I am proud to do so myself each year.

Secondly: My thanks must go to Dave Thomas, Chad Fowler, and Andy Hunt for their excellent work in the book “Programming Ruby 1.9 & 2.0” (aka the PickAxe book for its cover art). The fOOrth language is written in Ruby and without the three editions of that awesome book, my dead-ends, difficulties, and wasted effort would have been greatly multiplied.

Furthermore, the astute reader will not fail to observe that this document is very much fashioned in the style (if not the quality) of the PickAxe book. This imitation is my sincere flattery of the original.

Report Card

Naturally, it remains to be seen how well fOOrth is at actually accomplishing these lofty goals and living up to these principles. It may be imagined that at some later date, this guide could contain a “report card” examining this topic. For now, this matter is left as an exercise for the reader.

Update for V0.0.3

This version finds fOOrth with most of the core functionality and an initial draft user's guide and reference manual. This effort has taken a lot longer than was originally anticipated, but is now ready to proceed with incremental improvements.

11 Time after time, most people I speak to, assume that as a Ruby developer, I must really be a Rails developer.
12 See http://en.wikipedia.org/wiki/Main_Page

Update for V0.0.4

This minor version change was largely a test of the new Git branching model for development. Some notable code enhancements included:

- Fixes for class/subclass creation mode issues.
- The 2drop and 2dup methods.

Update for V0.0.5

Enough small changes accumulated to justify a step in version. Some notable changes:

- Support for nested contexts with no mode change.
- Array and Hash literals now run in the current mode rather than always deferred.
- Added the .empty method to Array and Hash; Added .length to Hash.
- Numerous fixes for Rational math. Conversions and rounding now more intuitive.
- Added the .join and .split methods to Array.
- Added missing documentation to the Thread class.
- Arrays and Hashes now display in fOOrth format. Previously Ruby formatting was used.

Update for V0.0.6

Re-factored the compiler sub-system.

Update for V0.0.7

Hot fix for a re-factoring bug and inadequate testing.

Update for V0.1.0

Significant changes: The introduction of Procedure Literals as an integral part of the compiling process. For linguistic harmony, methods like .each{ … } are now .each{{ … }} to match the way procedure literals work with {{ … }}. This change has resulted in a large reduction in the need for helper methods and other such kludges.

Update for V0.2.0

- Reworked the protocol of the .fmt methods and renamed them to format.
- Added the Mutex class.
- Added the Time class.

Update for V0.2.1

A minor update with a fix to method mapping, documentation upgrades and new Procedure methods call_v, call_x, and call_vx.

Update for V0.3.0

- Added the Duration class, a whole host of methods and documentation.
- Changed the Time class to return Duration objects when computing the span of two Time objects.
- The conventional format operation for all objects now has proper error handling added to catch malformed format strings.
- Minor assorted User Guide updates.
- Documented most of the System Call error types.

Installation

Ruby

The fOOrth system is written in Ruby, so, at this time, the first step in installing fOOrth is to install Ruby. The question that then arises is what version of Ruby is required? To date, fOOrth has been tested under:

- ruby 1.9.3p484 (2013-11-22) [i386-mingw32]
- ruby 2.1.5p273 (2014-11-13 revision 48405) [i386-mingw32][13]
- ruby 2.1.6p336 (2015-04-13 revision 50298) [i386-mingw32]
- Rubinius – to be tested!
- JRuby – to be tested too!

Given the versions that I know work, I am hopeful the 2.0.x and 2.2.x will likely work too. I can state with a great deal of confidence that fOOrth will NOT work with any MRI 1.8.x or older and 1.9.1 and 1.9.2 are very doubtful as well, but the confidence there is a lot less.

Installing Ruby is beyond the scope of this documentation, but some excellent references to assist in this endeavor are:

http://www.railsinstaller.org/en

Yes, this installer does install the Ruby and the Rails web framework, but includes comprehensive support for gem, git, devkit, rake, rdoc and other tools and is the easiest, most comprehensive choice for Windows or Mac (pre OSX Mavericks) users.

http://rubyinstaller.org/

Please note: Some changes in security may cause difficulty, so this work around may be helpful: https://gist.github.com/luislavena/f064211759ee0f806c88 or simply google the phrase “Workaround RubyGems' SSL errors”

Another comprehensive Ruby installation site for Windows and useful extensions not included above is:

https://www.ruby-lang.org/en/documentation/installation/

A comprehensive source for installing or even compiling Ruby on all platforms. This is also a hub of information on what is available in the world of Ruby.

Of course, fOOrth is built on Ruby, so it makes sense to understand the underlying foundation. For this this web site is invaluable: https://www.ruby-lang.org/en/.

13 Ruby 2.1.5 has serious code defects that are only resolved in 2.1.6. It is advised that the latter be used.

fOOrth

The fOOrth language is delivered in the form of a Ruby gem[14]. An easy-to-use package that delivers code with version management facilities. This gem is hosted on the web site Ruby Gems[15]. Once Ruby is installed, installing fOOrth is as simple as the following command:

```
gem install fOOrth
```

That's it! It should be that simple!

Installing from GitHub

GitHub[16] is the world's social network for programmers and above all, the code they create. With a GitHub membership (available for free), you get to rub shoulders with giants like Google[17] and Facebook[18]. To install fOOrth from its GitHub repository, some prerequisites must be met:

1. Git must be installed and working.
2. Ruby must be installed and working.

All of these requirements are met if you install via the RailsInstaller.org link above but there are numerous other ways to get these steps done. Otherwise, install Ruby (see above) and Git (https://git-scm.com/) from your favored sources.

Once the prerequisites are out of the way simply navigate you favorite web browser to:

https://github.com/PeterCamilleri/fOOrth

and click on the handy “Clone in Desktop” button and follow the instructions.

Contributing

Contributions to the fOOrth project should be made via GitHub. A summary of the recommended procedure for doing so follows:

1. Fork it
2. Switch to the development branch ('git checkout development')
3. Create your feature branch ('git checkout -b my-new-feature')
4. Do amazing things! Don't forget to write lots of tests!
5. Commit your changes ('git commit -am “Add some feature”')
6. Push to the branch ('git push origin my-new-feature')
7. Create a new Pull Request

14 This is not true. Until a reasonably stable code base exists the gem will not be available to avoid wasting a lot of repository space. Until then, the only way to get fOOrth is from github. See the source archive section below.
15 Ruby Gems is located at http://rubygems.org/
16 For a better introduction to GitHub please see: http://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1 or http://www.geekgumbo.com/2012/02/13/cloning-software-from-github/
17 Please see: https://github.com/google
18 Please see: https://github.com/facebook

It is strongly encouraged to apply all new coding efforts to the development (or a feature) branch and not master.

Plan B

Go to the GitHub repository and raise an issue calling attention to some aspect that could use some TLC or a suggestion or an idea. Apply labels to the issue that match the point you are trying to make. Then follow your issue and keep up-to-date as it is worked on. All input are greatly appreciated.

Running fOOrth

Once fOOrth is installed, there are several options for running the language environment. These are:

Rake:

From the base folder of the gem (where the file rakefile.rb is located) simply enter the following from the command prompt:

```
rake run
```

This will then run up an interactive, command line session in fOOrth.

Demo.rb:

In the base folder of the gem there is a file demo.rb. If the fOOrth gem has been installed on the local system, this can be copied to any convenient folder and run with:

```
ruby demo.rb
```

This will also run up an interactive, command line session in fOOrth. This method has the advantage of being a little quicker.

Testing

No code is well written that does not include a comprehensive set of tests and fOOrth is no exception. The tests in fOOrth are divided into two main sections: the unit tests which focus on testing the underlying Ruby support code and integration testing which takes a more wholistic approach and largely tests fOOrth with fOOrth code. To run these unit and integration tests, respectively, use the following commands:

```
rake test
rake integration
```

Known Issues

When testing under MRI Ruby 2.1.5, there is a known issue with the integration test suite. In particular, if the internal rake command that launches the test exceeds 1022 characters, the test will hang with no error. The only way to abort this error is to interrupt the test, typically by hitting a control-C character.

As a work-around, the names of the integration test files have been given a bit of a trim (the word library was shortened to lib) in order to limbo dance under the 1022 character limitation.

In the long run, there seems to be no fix for version 2.1.5 available. It may be that a switch to version 2.1.6 may be required to correct this issue properly[19].

Source Archive

The source code archive for fOOrth may currently be found on the github repository at the address: https://github.com/PeterCamilleri/fOOrth. See Appendix C – GIT for more details on the use of branching within the project.

19 Development is currently proceeding with Ruby 2.1.6.

First Steps

The fOOrth system can operate in a number of ways, but the classic mode is as an interactive programming environment. In this interactive system, the user enters commands in a text session. These are executed and any results appear as output to the screen. For example:

```
>4 5 + .
9
>
```

The “>” is a command prompt, the code entered was “4 5 + .” and the output was “9”. Drilling down a little deeper, fOOrth uses postfix notation, sometimes referred to as reverse polish notation[20] [21]. In more conventional languages, infix or algebraic notation is used. To add 4 and 5 we would write 4 + 5. The addition operator being infixed or between the operands. In postfix notation, the operands come first and the operator follows or is postfixed. Thus “4 5 +”.

Now infix algebra requires all sorts of complex operator precedence rules as well as parenthesis to override those precedence rules. Postfix notation needs no such complexity. Postfix notation is well supported by a simple stack[22] data structure and this is built into fOOrth in the form of its data stack.

Consider the example code in greater detail yet:

Tokens	**4**	**5**	**+**	**.**
Output				'9'
Data Stack	4	5	9	
		4		

Expressions that require parenthesis in infix notation, are written without them in postfix notation. Consider the following expressions:

Infix	Postfix	Result
2+3*4	2 3 4 * +	14
(2+3)*4	2 3 + 4 *	20

Note how the postfix version is simply read from left to right without having to jump about the expression and apply complex rules. So far, at this level, fOOrth appears to be identical to

20 http://en.wikipedia.org/wiki/Reverse_Polish_notation

21 The reason it is called Reverse Polish Notation (RPN) is that it is the reverse of the prefix notation created by Polish logician Jan Łukasiewicz (see http://en.wikipedia.org/wiki/Jan_%C5%81ukasiewicz), whose name is utterly unpronounceable by non-Polish speakers. I know. I tried. Even with Polish coaching, I never got it right.

22 http://en.wikipedia.org/wiki/Stack_(abstract_data_type)

FORTH. However this is not the case. To understand how this is so, consider the original addition of 4 and 5 at a deeper (but still abstract) level, contrasting the actions of fOOrth with a hypothetical FORTH implementation.

Language Tokens	Abstract Pseudo-Code Generated	
	FORTH	fOOrth
4	Push_integer 4	Push_integer 4
5	Push_integer 5	Push_integer 5
+	T_1 = Pop_integer T_2 = Pop_integer T_3 = Add_integers T_2 ,T_1 Push_integer T_3	T_1 = Pop_object T_2 = Pop_object T_3 = T_2.Add(T_1) Push_object T_3
.	T_1 = Pop_integer Print_integer T_1	T_1 = Pop_object T_1.Print_object

In FORTH, the + operator is hardwired as the integer addition operator. In fOOrth, the + operator is a message sent to a data item (or object) on the stack. The implementation of that operator is determined by how that object is programmed to respond to that message.

When incorrect data are sent to the FORTH “+” or the “.” words, they blindly proceed without regard for the incorrect results generated. FORTH has little to no error checking and usually handles errors by hanging or crashing. In fOOrth, each message that is sent, must be understood by its receiver. Errors are reported as soon as they occur.

The Syntax and Style of fOOrth

Syntax

In most programming languages most of the outline of the code and its major control structures is a function of the syntax enforced by the parser. In languages like Smalltalk, FORTH, and fOOrth this is not the case. The parser only supports the barest essentials required to create expressions, the rest is a consequence of the actions taken by those expressions. In FORTH, the only constructs recognized by the parser are code “words” and numeric literals. In fOOrth, a bit more is done with the parser supporting “words” (aka “methods”), numeric literals, string literals, and comments. Thus syntax plays a minor role in fOOrth. It is more semantic than syntactic in nature.

Spaces

In general, fOOrth language tokens or words are separated by spaces. Other languages allow operators and punctuation to abut identifiers and literal values. In fOOrth this is generally not permitted. Thus

```
4 5 + .     // Correct, prints out 9

4 5+.       // Incorrect, produces the error F10: ?5+.? as in what's this?
```

In fOOrth, there are three exceptions to this rule: Comments, String literals and Numeric literals.

Comments

The fOOrth language supports two types of comments. Embedded comments may be placed inline between other language elements as in this silly example of how not too add comments to your code:

```
4 (four) 5 (five) + (add) . (print)
```

Note that unlike FORTH, there is no need to place a space after the leading “(“ character. The other form of comment is lifted from C++ and is started a “//” and ends at the end of the line.

```
4 5 + .           // Prints out 9!
```

While this example has an extra space after the //, this is not required.

String Literals

In fOOrth special support is provided for embedded string literals. Any fOOrth method that ends with a " character is assumed to contain an embedded string. No space is required between the the " and the start of the string. The string ends with a matching trailing " character. Some examples of methods with embedded strings are:

```
."Hello World"    // Print Hello World

)"ls -al"         // Shell out the command: ls -al

"ABCD"            // The string literal "ABCD"
```

Further discussion of string literals is found in section String – String Literal below.

Numeric Literals

Many sorts of numeric literals require various sorts of punctuation as part of the number being specified. These as placed inline with no spaces as in these examples:

```
-10   99.1  -3.0E21   1/3   2+3i  -2-2i
```

Further information on these literals is found in the sections “Complex”, “Float”, “Integer”, and “Rational”.

Procedure Literals

In fOOrth special support is provided for embedded procedure literals. Any fOOrth method that ends with a {{ character sequence is assumed to contain an embedded procedure or code fragment. A space *is* required between the the {{ and the start of the code in the procedure. The string ends with a matching trailing }} sequence. A space is needed there too, between the end of the code and the }}.

Some examples of methods with embedded procedures are:

```
{{ dup + }}

array_value .map{{ v dup * }}

"name" Outstream .append{{ ~"Hello" }}
```

A fOOrth Calculator

Since fOOrth was in part inspired by the powerful RPN calculators manufactured by companies like Hewlett Packard, let's start delving deeper into fOOrth using its interactive mode as a kind of super-calculator.

The Basics

To begin, run up fOOrth (see section Running fOOrth above). Lets see what comes up initially.

```
C:\Sites\fOOrth>rake run
Welcome to fOOrth: fO(bject)O(riented)rth.

fOOrth Reference Implementation Version: 0.0.3

Session began on: 2015-01-24 at 11:39am

>
```

The last line has a “>” which is a prompt for input. To facilitate this use of the language, we now enter the)show command.

```
>)show

[ ]
>
```

For clarity, user input is shown bold and underlined. This emphasis is for clarity in text only, and does not occur in the actual interactive session.

Also note the [] after the command. This is a data dump that will help us keep track of the contents of the stack. No data is shown between the brackets because at this time the data stack is empty. Lets dive right in and try some calculating:

```
>4 5 6

[ 4 5 6 ]
>*

[ 4 30 ]
>+

[ 34 ]
>.
34
[ ]
```

In the above, the numbers 4 5 6 are entered to the stack, then 5 and 6 are multiplied, then 4 and 30 are added. Finally, the result, 34, is printed out to the screen (with the “.” dot command), leaving an empty stack once more.

Naturally, computations are not limited to integer values, but may include floating point, rational and even complex data. Some example computational sequences with these data types are show in the next screen capture sequence:

```
>4.0E3 50.0 6000.0

[ 4000.0 50.0 6000.0 ]
>*

[ 4000.0 300000.0 ]
>+

[ 304000.0 ]
>.
304000.0
[ ]
>1/2 2/3 4/5

[ 1/2 2/3 4/5 ]
>*

[ 1/2 8/15 ]
>+

[ 31/30 ]
>.
31/30
[]
>1+2i 3+4i 5+6i

[ 1+2i 3+4i 5+6i ]
>*

[ 1+2i -9+38i ]
>+

[ -8+40i ]
>.
-8+40i
[ ]
>
```

The fOOrth language supports many common math operations. In addition to the four basic operators (add +, subtract -, multiply *, and divide /) there are (remainder mod, exponentiation **) as well as trigonometric, logarithmic and other operators too numerous to mention. Most will be found in the class reference section for the Numeric class.

Stack Manipulation

All RPN calculators (and even most non-RPN ones) include a number of operations for manipulating the stack. In fOOrth, these operations are pretty much lifted verbatim from FORTH. These operations are performed directly by the Virtual Machine, again, just like FORTH would have done. Here are some brief examples of the most common sorts of operations plus a look at them in action interactively:

drop – discard the top element of the stack.

```
>1 2 3

[ 1 2 3 ]
>drop

[ 1 2 ]
```

dup – duplicate the top reference or value. Note that any referenced data is NOT duplicated. See the section Cloning Around for further details.

```
>"apple"

[ "apple" ]
>dup

[ "apple" "apple" ]
```

nip – grab the second element of the stack and discard it.

```
>1 99 2

[1, 99, 2]
>nip

[1, 2]
```

over – grab the second element of the stack and push a duplicate of it to the top of the stack.

```
>"apple" "pie"

[ "apple" "pie" ]
>over

[ "apple" "pie" "apple" ]
```

pick – pick the stack element indexed by the top stack element and make a duplicate of that the new top element of the stack.

```
>1 2 3 4

[ 1 2 3 4 ]
>2 pick

[ 1 2 3 4 3 ]
```

swap – exchange the top two elements of the stack. Just like the old x↔ y calculator key

```
>1 2

[ 1 2 ]
>swap

[ 2 1 ]
```

tuck – take the top element of the stack and tuck a duplicate of it under the second element.

```
>1 2
[ 1 2 ]
>tuck

[ 2 1 2 ]
```

The Return Stack

The astute reader and scholar of FORTH will note the absence of the FORTH return stack. Quite simply, the return stack is not necessary in fOOrth and would serve little purpose. Early versions of fOOrth did indeed have such a stack, but it has been a long time since it was utilized in anyway. In classical FORTH, the return stack serves three purposes:

1. To store return addresses for word calls. In fOOrth this is handled by the Ruby virtual machine.
2. To store context for control structures. In fOOrth the context mechanism provides for a far richer and more reliable set of control, compile, and data structures.
3. As an escape valve when the stack has become too crowded and an need to put data “someplace” brings the return stack into service. In fOOrth local and instance variables provide a much more flexible and less error prone alternative. In addition, the object oriented concept of “self” often allows for a great deal of code simplification.

Programming

The very best calculators not only excelled at computations, they also allowed actions to chained together and stored. They were programmable. This feature greatly extends the reach of these devices. The fOOrth language calculator is eminently programmable.

Let us start with the simplest form of programming, the creation of virtual machine methods. This closely corresponds to the creation of “words” in FORTH. As an example lets us create a very simple method called double that doubles a value. The transcript follows:

```
>: double dup + ;

[ ]
>4 double .
8
[ ]
>"apple" double .
appleapple
[ ]
>
```

The colon is used to start a definition on the virtual machine. This is followed by the name of the method, in this case “double”. The body of the definition follows and finally the semi-colon closes off the definition. This is all classic FORTH code.

When we enter **`4 double .`** we get an answer of 8, just as expected. The differences to classic FORTH begin to show when we enter the **`"apple" double .`** command. Instead of an error or some crazy number, or a crash, we get the string “appleapple”. The double method “doubled up” the string.

This is a result of the fact the the + operator in the double method is not hardwired to integer addition as it would be in FORTH, but is sent to the receiver where it is processed according to the rules of that receiver. For an integer, that is integer addition. For a string that is string addition, usually called concatenation.

Control Structures

Now recording programming steps is all well and good, but any decent calculator also has the ability to make decisions and perform repetitive tasks, and fOOrth does not disappoint!

The if statement:

While a calculator might have settled for a conditional “goto” statement, fOOrth has a fully structured “if” statement. It is in RPN however so the Boolean expression comes before the “if” operator as in this example:

```
>: is_five 5 = if ."It is five!" else ."Nope, it is not five" then ;

>4 is_five
Nope, it is not five
>5 is_five
It is five!
```

In this example, a method called is_five is created that takes different actions based on a test of the input argument. The “if” statement defines two local methods, “else” and “then”. A more formal look at the “if” statement is:

<boolean expression> if <true clause> {else <false clause>} then

Where the { } indicate an optional component.

The switch statement

Now, fOOrth, FORTH and Smalltalk share a shortcoming. They all have difficulty dealing with chained if then elsif elsif elsif end situations. They tend to cascade many levels of nesting inside the “else” clauses. For a real example of this the following code is presented. This ugly_if[23] code uses nested “if” statements to select from three choices with a default.

```
// ugly_if.foorth – ugly nested if statements
: choose_path
    dup 1 = if
      drop ."path 1"
    else
      dup 2 = if
        drop ."path 2"
```

23 The file is ugly_if.foorth which may be found in the docs/sippets folder.

```
    else
      dup 3 = if
        drop ."path 3"
      else
        drop ."Invalid path selected"
      then
    then
  then
  cr ;
```

Note the “creeping” indenting of the code, a reflection of the nesting of the control structures. To alleviate this problem fOOrth has the switch construct. Consider instead, the following snippet[24] of code:

```
// switch.foorth -- switch statement sample
: choose_path
  switch
    dup 1 = if drop ."path 1" break then
    dup 2 = if drop ."path 2" break then
    dup 3 = if drop ."path 3" break then
    drop ."Invalid path selected"
  end  cr ;
```

The purpose of the switch … end control structure is to group together a number of statements. In addition to the “end” keyword, the switch clause defines the “break” verb. The purpose of the break (and its related ?break) verb is to skip past any remaining statements to the just past the “end”.

When run (both versions produce the same output, but switch.foorth is shown in this example) we see:

```
>)load"docs/snippets/switch.foorth"
Loading file: docs/snippets/switch.foorth
Completed in 0.0 seconds

>1 choose_path
path 1

>2 choose_path
path 2

>42 choose_path
Invalid path selected
```

The do statement

Another major feature of a good programmable calculator is the ability to automate repetitive operations. In fOOrth, the “do” statement borrows heavily from FORTH, but there are some crucial differences. A classic snippet of code might look like this:

```
>0 10 do i . space loop
0 1 2 3 4 5 6 7 8 9
```

24 The file is switch.foorth which may be found in the docs/sippets folder.

This prints out the numbers from 1 through 9 to the terminal session. The “do” and “loop” commands mark the boundaries of the loop; The “i” command is used to retrieve the current loop counter value. For nested loops, the “j” command retrieves the value of the outer loops counter value. By default, the “loop” command adds one to the loop value. For more flexible looping, the “+loop” command allows an arbitrary increment value to be specified.

So far, fOOrth “do” loops are just like FORTH. However, a significant difference between fOOrth and FORTH is how the end condition is determined. In FORTH, the loop terminates when the current loop value is exactly equal to the end value. In fOOrth, the condition is tripped when the current loop value is greater than or equal to the end value. An example of this in action is the following broken code:

```
>10 0 do i . space loop
```

In fOOrth, this code does nothing because the end condition is met at the start of the loop. In FORTH it goes looping off crazily until the loop counter overflows and counts up to zero. That can be a very long time indeed and is as close to an infinite loop as does not matter.

-i and -j

In solving one problem, often a new problem is created, and this is no exception. The astute reader will be wondering how a loop would be constructed that counts *backwards*! The broken code above does nothing and so does the classical way of reverse counting in FORTH:

```
>10 0 do i . space -1 +loop
```

To resolve this issue, fOOrth provides reverse counter versions of the loop variables. The reverse counter for “i” is “-i” and the reverse counter for “j” is “-j”. Thus the fOOrth version of this code is simply:

```
>0 10 do -i . space loop
9 8 7 6 5 4 3 2 1 0
```

Now both the forward and reverse loop variables are available so it is possible to process *both* directions at once in a single loop. For example:

```
>0 10 do i -i * . space loop
0 8 14 18 20 20 18 14 8 0
```

A simple example of looping in action can be seen in the times_table[25] example file. Here is the source code:

25 The file is times_table.foorth which may be found in the docs/sippets folder.

```
// Print out a classic times table.

cr
." *  |"  1 13 do i f"%3d " .  loop cr
."----+" "-" 47 * .  cr

1 13 do
  i f"%2d  |" .

  1 13 do
    i j * f"%3d " .
  loop

  cr
loop
cr
```

Most of this code is devoted to making the output look nice, but the core of the code are the nested do loops both counting from 1 to 12. As can be seen, the inner loop counter is accessed via the “i” method and the outer counter is accessed via the “j” method. And here is the output!

```
>)load"docs/snippets/times_table"
Loading file: docs/snippets/times_table.foorth

 *  |  1   2   3   4   5   6   7   8   9  10  11  12
----+------------------------------------------------
 1  |  1   2   3   4   5   6   7   8   9  10  11  12
 2  |  2   4   6   8  10  12  14  16  18  20  22  24
 3  |  3   6   9  12  15  18  21  24  27  30  33  36
 4  |  4   8  12  16  20  24  28  32  36  40  44  48
 5  |  5  10  15  20  25  30  35  40  45  50  55  60
 6  |  6  12  18  24  30  36  42  48  54  60  66  72
 7  |  7  14  21  28  35  42  49  56  63  70  77  84
 8  |  8  16  24  32  40  48  56  64  72  80  88  96
 9  |  9  18  27  36  45  54  63  72  81  90  99 108
10  | 10  20  30  40  50  60  70  80  90 100 110 120
11  | 11  22  33  44  55  66  77  88  99 110 121 132
12  | 12  24  36  48  60  72  84  96 108 120 132 144

Completed in 0.21 seconds
```

Now we're all ready for those math tests!

The begin statement

The “do” loop is great for cases where the iteration action is based on counting. For loops *not* based on counting there is the “begin” statement. The “begin” keyword is balanced against one of the following locally defined terminating keywords:

- begin … until – loops until the top of stack is true.
- begin … again – loops indefinitely
- begin … repeat – same as above.

Now it will be noticed that two of the configurations loop indefinitely. This is not desirable, so to handle this case the “while” verb exists. The “while” method exits the loop if the top of stack is false. A begin … {until/again/repeat} loop may have multiple while sub-clauses.

A simple (simplistic) example of this type of loop in action is seen in the int_log2[26] snippet:

```
// A simple integer log2
: ilog2 .to_i 2/ 0 swap
  begin
    dup 0> while
    2/ swap 1+ swap
  again
  drop ;
```

A sample run is shown below:

```
>)load"docs/snippets/int_log2"
Loading file: docs/snippets/int_log2.foorth
Completed in 0.005 seconds

>8 ilog2 .
3
>18 ilog2 .
4
>0 ilog2 .
0
>1 ilog2 .
0
>3 ilog2 .
1
>4 ilog2 .
2
```

Data Memory

Even the most rudimentary calculators are equipped with some data storage, even if it is the primitive STO, RCL, M+, M-, and MCLR. In real programming systems, data storage is rather more complex, more than can fit into this already too long section. In fOOrth, considerable flexibility exists in the use of data memory. The following section examine this topic in several categories.

26 The file is int_log2.foorth which may be found in the docs/snippets folder.

Data Storage in fOOrth

Typing

While the data itself is strongly typed in fOOrth, the data storage (variables etc) are not. Data of any sort may be placed into a variable. This closely reflects how Ruby does things. FORTH in contrast is a completely type less language with no type checking at any point or on any level.

All of this begs the question though: What is a data type? In fOOrth, data types are compatible if they respond to the required set of messages and produces the expected results. This is covered in more detail later, but for now we can simply say that the type of a datum is determined by the operations it supports. This is often called Duck Typing[27]. Again, fOOrth borrows heavily from Ruby.

Declarations

In Ruby, there are no formal declarations of variables. There are times when the extreme flexibility of Ruby forces the programmer to write a preemptive assignment statement to force the language to do the right thing, but there are no variable declarations[28]. In fOOrth, variables are always declared and they are always given an initial value. The general form of one of these declarations is:

```
<value> <defining_word:> <variable_name>
```

The details of this declaration are filled in by the following sections.

Scoping

In all programming languages, variables have a life span, or scope of existence. The fOOrth language supports four scoping options. These are described below:

Local Scope

The local scoping option allows variables to exist locally within a single method. Typically, local variables are created near the beginning of the method, their initial values may be literals, computed values or may be taken from arguments to the the method.

Methods: `val:` and `var:`

Regex for valid local variable names: `/^[a-z][a-z0-9_]*$/`[29]

Notes: Local variables are only accessible inside the methods they are defined in, after the point in the code where they are defined.

---

27 From the adage: If it quacks like a duck, swims like a duck, and waddles like a duck... it's a duck!

28 Many think that the attr_reader, attr_writer, and attr_accessor macros of Ruby are variable declarations, but they are not. They simply define access methods for a variable, not the variable itself.

29 See Appendix B for more information on Regular Expressions.

Instance Scope

Instance scoped variables are associated with instances of objects. As such, these methods are only accessible inside methods of those objects. Instance variables are distinguished by the leading "@" sign in their names. Instance variables can only be created in such methods as well, with the `.init` method being the most popular since it is called to initialize a new instance of the object.

Methods: `val@:` and `var@:`

Regex for valid local variable names: `/^@[a-z][a-z0-9_]*$/`

Notes: Instance values/variables are only accessible in environments where the "self" entity is the object where they exist. This is the case of a method or a where{...} clause of that object.

Thread Scope

Thread variables are associated with the thread in which they are defined. As such they are accessible anywhere within that thread. Thread variables are distinguished by the leading "#" sign in their names. Thread variables may be created at any point within the thread.

Methods: `val#:` and `var#:`

Regex for valid local variable names: `/^#[a-z][a-z0-9_]*$/`

Notes: When a new thread is created, it receives a copy of the thread variables in the thread that created it.

Global Scope

Global variables may be accessed at any point after they have been defined. Global variables are distinguished by the leading "$" sign in their names.

Methods: `val$:` and `var$:`

Regex for valid local variable names: `/^\$[a-z][a-z0-9_]*$/`

Notes: Global variables are considered bad in many circles.

Referencing

This one is a bit tricky. Most programming languages have the concept of the value *of* a variable and a reference *to* a variable. In "C", documentation speaks of "lvalues" and "rvalues". These labels describe the role (left and right value) played in the classic "C" assignment statement:

```
<lvalue> = <rvalue>;
```

In "C" there is the further concept of being able to create a reference to a variable using the "&" operator. This operator allows the programmer to create a reference (via a pointer) to a the variable in question.

Ruby on the other hand has no explicit support for references. Variables themselves are always

values and there exists no way to generate a reference to a variable. It is true that the Ruby interpreter must have access to a reference to a variable in order to perform an assignment, but this capability is kept locked up in the internals of the language.

In fOOrth, the ability to use references and values is explicitly available to the programmer through the “var” and “val” keyword roots[30]. To create a variable that holds a reference, use:

```
<value> var: <var_name>
```

For example:

```
0 var: score
```

To create a value simply substitute the var: version as in this example:

```
10 val: max_score
```

The first example creates a variable “score” that is a reference to the value, currently 0. The second creates a variable “max_score” with a value of 10. Next we examine how referencing affects the code that is needed to use these variables:

Task	*var*	*val*
Sample Declarations	0 var: score	10 val: max_score
Just what is being declared?	A method (called score) that pushes a reference to the value (0) onto the stack.	A method (called max_score) that pushes the value (10) onto the stack.
Retrieve the value of the variable.	score @	max_score
Update the value of the variable.	1 score !	--[31]
Get a reference to the variable.	score	--[32]

As can be seen in the above table, var scope is more capable than val scope. It is also slower, more complex, and more verbose. For most uses, the greater capabilities of the var scope are not required. Thus it is expected that for most applications val scope will be the predominant form utilized.

In most programming languages, including Ruby, val or value variables are called constants. In Ruby, this title is a falsehood due to the issue of data mutation, covered in the next section.

Mutation

In motion pictures, mutants are sometimes the good “guys”, but regardless of that, wherever mutations are involved, trouble always seems to follow them. That also holds true for fOOrth

30 For simplicity, the examples here assume local scope, but the examples would work in the same manner with global, thread, or instance scoping.
31 This operation is not available for value variables.
32 This operation is also not available for value variables.

and the underlying Ruby base language. In fOOrth, all datum are divided into two major classifications: Immutable and Mutable. Simply put, immutable values are those that retain their value when operations are applied to them. Mutable values do not have this property. In fOOrth, numbers (of all sorts), boolean values (true and false) and the special value nil, are all immutable. Everything else *is* mutable. It is noteworthy that character strings in particular fall into the mutable camp.

For comparison, consider this first example with immutable data:

```
>5 val$: $iv   $iv .
5
>$iv 6 + .
11
>$iv .
5
>
```

In this example, a value of 5 is created. An addition operation with 6 is performed, yielding 11. Nonetheless, the original value of 5 is NOT mutated by this operation. Now consider a similar scenario with mutable data:

```
>"Hello" val$: $mv  $mv .
Hello
>$mv " World" << .
Hello World
>$mv .
Hello World
```

In this case, the string variable IS mutated by the concatenation “<<”, operator. If one where relying on the $mv value to be constant, this would be a severe setback. Now to be clear, there is a non-mutating concatenation operator, “+”. As shown below, it does NOT mutate the string:

```
>"Hello" val$: $mv  $mv .
Hello
>$mv " World" + .
Hello World
>$mv .
Hello
>
```

So why have both? Why not always avoid the mutation? Simply put, the non-mutating version is slower because it must create a copy of the string to avoid modifying the original. There is a trade-off between mutation and efficiency. With immutable data, there is no need for trade-offs or two versions of operations. Operations on immutable data are always immutable AND efficient!

The fOOrth language does provide ways to explicitly control or at least work around mutation issues. This is discussed in the next section on Cloning Data.

Cloning Data

Since some data in fOOrth are mutable, it is sometimes necessary to create copies of that data so that operations can be performed that do not corrupt the “original” data. The fOOrth language system has a number of data duplication methods that meet various needs. These methods are summarized below:

Method	Stack Before	Stack After	Description	Time Used	Copy Depth
dup	x	x x	Duplicate the data without any copying.	Least	None
copy	x	x x'	Duplicate the data with a shallow copy.	Moderate	One Level
.copy	x	x'	Replace the data with a shallow copy.		
clone	x	x x″	Duplicate the data with a deep copy.	Most	All Levels
.clone	x	x″	Replace the data with a deep copy.		

Deep vs Shallow Copy

To examine the differences in the copying strategies, consider the following three different scenarios:

1. No Copying
2. Shallow Copying
3. Deep Copying

No Copying

In the following example session, the first line creates a value, “$expo” with an array as value. The first element of the array is the string “Expo” and the second element is the number 67. To show this, the value is printed out. The second statement creates a value “$ecopy” with the same value as “$expo”. The next two statements[33] change the 67 to a 99 and append the string “sure” to the string “Expo”. The final two statements display “$ecopy” (the copy) and “$expo” (the original).

```
>[ "Expo" 67 ] val$: $expo  $expo .
[ "Expo" 67 ]
>$expo val$: $ecopy  $ecopy .
[ "Expo" 67 ]
>99 1 $ecopy .[]!

>0 $ecopy .[]@ "sure" <<

>$ecopy .
```

33 For more information on array access words, please see the Array section below.

```
[ "Exposure" 99 ]
>$expo .
[ "Exposure" 99 ]
```

As can be seen, both have been modified in the same way because the two values (“$expo” and “$ecopy”) both reference the same mutable array.

Shallow Copying

This session is the same as the previous except that the “.copy” method is applied to the value before it is used to create “$ecopy” The “.copy” creates a copy of the array but not the data elements in that array.

```
>[ "Expo" 67 ] val$: $expo  $expo .
[ "Expo" 67 ]
>$expo .copy val$: $ecopy  $ecopy .
[ "Expo" 67 ]
>99 1 $ecopy .[]!

>0 $ecopy .[]@ "sure" <<

>$ecopy .
[ "Exposure" 99 ]
>$expo .
[ "Exposure" 67 ]
```

As can be seen, the results are mixed. Since a copy of the array was made, the number 67 is not changed in the original. The string, “Expo” however is still mutated to “Exposure” since no copy of it was made. Of course we could have used the non-mutating version of string concatenation. This would have resulted in the following, somewhat longer code:

```
>0 $ecopy .[]@ "sure" + 0 $ecopy .[]!
```

Since the string is not being mutated, but instead, a new string is being created, it is necessary to store this new string back into the array explicitly.

In this new statement, the “`0 $ecopy .[]@`” retrieves the existing string, the “`"sure" +`” performs the non-mutating concatenation, and the “`0 $ecopy .[]!`” stores the string value just computed back into the array.

Note: For simple mutable data like strings, this shallow copy is fully sufficient to protect against any unwanted data mutation. However, for more complex data with multiple levels of information (like arrays, hashes, or user defined classes) a more thorough copying method is needed.

Deep Copying

In the final example session, the '.copy” is replaced with “.clone”. This performs a deep copy that copies the array and its contents (and any of the contents' contents etc, etc...[34]).

```
>[ "Expo" 67 ] val$: $expo  $expo .
```

34 From “The King and I”, please see http://en.wikipedia.org/wiki/The_King_and_I

```
[ "Expo" 67 ]
>$expo .clone val$: $ecopy  $ecopy .
[ "Expo" 67 ]
>99 1 $ecopy .[]!

>0 $ecopy .[]@ "sure" <<

>$ecopy .
[ "Exposure" 99 ]
>$expo .
[ "Expo" 67 ]
```

As can be seen, the original value “$expo” is not modified in anyway by changes made to its clone in “$ecopy”.

Partial Copying

The clone commands in fOOrth have allowance for the fact that it is not always desirable to copy all of the data members when a clone is made. This is handled with the “.clone_exclude” method. Any object that defines this method is able to exclude certain data from the copying process.

If the “.clone_exclude” is defined as a shared method for a class, then all instances of that class share the same exclusions. On the other hand if it is defined exclusively for a single object, only that object is affected.

The “.clone_exclude” method returns an array of items to be excluded from the copying process. For most objects, these are the names of instance variables as strings. For arrays and hashes, these are the specific index values to be skipped over during the copying. Note that if the named variables or index values do not occur in the object being cloned, no action is taken. Some examples follow:

```
(Clone exclusions in a class.)
class: MyClass
MyClass .: .init "a" val@: @a "b" val@: @b ;
MyClass .: .clone_exclude [ "@b" ] ;
```

In the above, when instances of MyClass are cloned, the variable @a will also be cloned, while @b will not. The @b variable will be shared between the originals and the clones.

```
(Clone exclusions in an array.)
[ "apple" "banana" ] val$: $test_array
$test_array .:: .clone_exclude [ 1 ] ;
```

In this case, when the array is cloned, the contents of the array except for location 1 will also be cloned. The contents of position 1 will be shared between the originals and the clones.

```
(Clones exclusions in a hash.)
{ 0 "apple" -> 1 "banana" -> } val$: $test_hash
test_hash .:: .clone_exclude [ 1 ] ;
```

Again in this case, the contents of the hash except for the entry indexed by 1 will also be cloned. The contents of index 1 will be shared between the originals and the clones.

Permissive Copying

In Ruby, if an attempt is made to clone an immutable data item like a number, an error occurs. The justification for this uncharacteristic strictness are not at all clear, but it means that the clone operation must be applied with great care.

In fOOrth, this is not the case. When clone or copy are applied to immutable data, the data is returned without modification or error. The reasoning here is simple. The data in question are already immutable so nothing needs to be done. Doing nothing is *not* an invalid operation. So fOOrth does precisely that and the program continues without error.

Handling Exceptions

Error handling is an essential, if unpopular, part of all programming tasks. This section focuses on the use of the exceptions mechanism to simplify and streamline error handling.

When exceptions are not employed, code must be written that detects an error and returns an "error code". Further code must then be written to detect these error codes and either process them, or propagate them up the call chain until they can be handled at the appropriate level. In this approach, it is not uncommon for error handling code to be so voluminous that it obscures the "main" flow of the code.

Exceptions are an error handling mechanism designed to separate out error handling from error detection and to simplify code structures. When errors are detected, they can be "thrown". No error codes need be returned. Exceptions are simply "caught" and processed at the appropriate level. The exception system handles the propagation of errors without the need to write additional code.

The Nature of Exceptions in fOOrth

Exceptions is one area where fOOrth is very different than Ruby[35]. In Ruby, exceptions are objects that are part of an elaborate hierarchy of exception classes, one class for each type of exception. In fact, there are so many exception classes that the majority of classes in the Ruby environment are exception classes. In fOOrth exceptions are simply messages sent from the error detector to the error processor. These messages take the form of a string with a leading part that is structured and a trailing part that is free form and hopefully descriptive.

Consider a typical exception message in fOOrth:

```
E15: divided by 0
```

The two components of this message are clearly visible. The structured section consists of "E15:" and identifies the exception. The unstructured section "divided by 0" describes the nature of the exception. The use of these sections is examined further under the topic "Determining the Type of Error", below.

Handling Errors

Consider the following three methods[36] that display 50/(n-5) for values of n from 0 to 9:

```
(Exceptions in action)
(Phase One - Living Dangerously)
: danger
  0 10 do
    50 i 5 - / . space
  loop ;
```

35 Even so, fOOrth exceptions are implemented using the Ruby exception mechanisms.
36 These examples are in the file exception.foorth in the docs/snippets folder.

```
(Phase Two - Living Tediously)
: tedium
  0 10 do
    50 i 5 -
    dup 0<> if
      / . space
    else
      drop drop ."oops "
    then
  loop ;

(Phase Three - Living Exceptionally)
: safety
  0 10 do
    try
      50 i 5 - / . space
    catch
      drop ."oops "
    end
  loop ;
```

The first method called “danger” ignores errors totally[37]. It just runs without a care because, surely, what could possibly go wrong? The second method called “tedium”, adds some extra code to detect a possible error condition (like division by zero) and print out a helpful message, “oops”. The last method called “safety” contains the potentially dangerous code in a “try” clause. If any error should be detected, the “catch” clause is executed which also prints out the vitally important “oops” message.

Let's see what happens when we try out these three methods...

```
>danger
-10 -13 -17 -25 -50
E15: divided by 0
>tedium
-10 -13 -17 -25 -50 oops 50 25 16 12
>safety
-10 -13 -17 -25 -50 oops 50 25 16 12
>
```

The living dangerously (danger) code bombs out with an error. Not good. The results from the tedious (tedium) and exceptional (safety) code are the same. The difference is that the exceptional code is clearer and more concise and the tedious code is well... tedious.

The basic try block

The bare basics of an exception handled code block is:

```
try (dangerous code here) catch (error recovery code here) end
```

Note that the dangerous code may include nested method calls, control structures, etc.

37 Well it's more complex than that. It is more accurate to say that when exceptions occur and no handler is found, the default exception handler is executed which takes safe, default actions that may or may not desirable to the correct operation of the program that had the error.

Furthermore these entities must be complete. You cannot have part of a loop or conditional statement. That would generate an error at compile time.

Determining the kind of error

In fOOrth, the catch clause catches all errors[38]. Often, different corrective action is required depending on the type of error. To determine the type of error, fOOrth uses the error code prefix. This prefix is a string consists of a leading upper-case letter, followed by a two digit code, optional followed by a comma and a sub-code, finally ending with a colon (":").

The defined error codes are specified below in the sections: fOOrth Native Exception Codes, Application Error Codes, and Ruby Mapped Exception Codes.

To facilitate the checking of error codes in the catch clause, the local method ?" is used. This method has an embedded string that is matched against the current error to see if there is a match. Consider the case of the divide by zero error from the previous examples. The following statements will all test for this error:

```
(stuff omitted) catch ?"E" if (action) then
(stuff omitted) catch ?"E1" if (action) then
(stuff omitted) catch ?"E15" if (action) then
(stuff omitted) catch ?"E15:" if (action) then
```

Whereas the following would NOT process that error:

```
(stuff omitted) catch ?"F" if (action) then
(stuff omitted) catch ?"E2" if (action) then
(stuff omitted) catch ?"E**" if (action) then
(stuff omitted) catch ?"E15,12" if (action) then
```

So far, the examples have focused on handling a single type of exception using an if statement. A more general approach utilizes the switch statement(see Control Structures, the switch statement above for more details). An example follows:

```
try
  (dangerous code)
catch
  switch
    ?"F30:" if (action 1) break then
    ?"E15:" if (action 2) break then
    bounce (See Passing the buck, below)
  end
end
```

Passing the buck

Given that many types of exceptions exist, it will naturally occur that an exception handler will probably not handle all possible conditions. To deal with this situation, the "bounce" verb allows

38 This is not actually true. There are some errors like "fatal" that are not caught because they are in effect not recoverable and catching these errors and trying to recover would lead to even worse problems. Other missed exceptions are simply gaps in the implementation and will eventually be handled correctly in a later version of the fOOrth language system.

an exception handler to relaunch or bounce the exception to the exception handler at the next higher level in the call chain. The last example in the previous section shows this in action.

The handler processes exceptions of type F30 and E15 itself. All other types of exceptions are bounced up the call chain.

Tying Up Loose Ends

An important aspect of programming is the management of resources. A large part of that task involves freeing up, closing, deleting, or otherwise retiring objects used in by the program. While useful, exceptions can bypass this clean-up work. Avoiding this problem is the reason for the “finally” keyword.

In a try block, the finally section represents code that is performed after the dangerous code runs, regardless of the success of that code. The finally section always gets the last word. Consider this fourth[39] method in the exceptions[40] file:

```
(Phase Four - Cleaning Up After Yourself)
: cleanup
    "temp.txt" OutStream .create val: out_file
    ."File opened" cr

    try
      ."Danger comes next." cr
      1 0 / out_file .
      ."Danger has passed." cr
    finally
      out_file .close
      ."File closed" cr
    end ;
```

This method opens a file, tries to write the result of a dangerous calculation to it, and then, finally, closes the file. Along the way, the chatty code gives progress reports so that we can follow its progress in this perilous task. So, let's see what happens when this code is run.

```
>)load"docs/snippets/exception.foorth"
Loading file: docs/snippets/exception.foorth
Completed in 0.02 seconds

>cleanup
File opened
Danger comes next.
File closed

E15: divided by 0
```

The file is opened, upcoming danger is announced but never passed, and then the file is closed. We then see the default exception handler telling us what went wrong. The key here is that the file was closed even though an error was encountered.

39 That's fourth and not FORTH.
40 This example is in the file exception.foorth in the docs/snippets folder.

Summary

The try block brings all the elements discussed in the previous sections as laid out below. Note that the order of sections is important. A catch section cannot follow a finally section.

```
try
  (dangerous code)
(optional) catch
  (exception handler with optional ?"Err Code" and bounce)
(optional) finally
  (cleanup code goes here)
end
```

The last example in our file[41], shows all of these sections working together:

```
(Phase Five - All Together Now)
: last_example
    "temp.txt" OutStream .create val: out_file
    ."File opened" cr

    try
      ."Danger comes next." cr
      1 0 / out_file .
      ."Danger has passed." cr
    catch
      drop
      ."Error detected." cr
    finally
      out_file .close
      ."File closed" cr
    end ;
```

And here is the output for this code:

```
>last_example
File opened
Danger comes next.
Error detected.
File closed

>
```

Note how the error is caught (displaying “Error detected”) and then cleanup actions are performed (displaying “File closed”). Unlike the fourth[42] method, there is no uncaught exception and the default exception handler is not utilized. Instead the example exits gracefully.

41 This example is in the file exception.foorth in the docs/snippets folder.
42 Still not the FORTH method.

Generating Errors

So far our code has been responding to errors and handling them as needed. The question arises: What if we need to take on the role of whistle-blower[43] when we detect an error? In fOOrth, exceptions are messages sent from the detector to the handler. So, just as strings are caught to handle exceptions, they are thrown to generate them.

Consider the following security testing code[44]:

```
(Is the password secure?)
: test_password (password -- )
  "1234" = if
    throw"U10: Change the combination on my luggage!"
  then ;
```

When run we get:

```
>)load"docs/snippets/throw.foorth"
Loading file: docs/snippets/throw.foorth
Completed in 0.01 seconds

>"1234" test_password

U10: Change the combination on my luggage!

>"secret" test_password

>
```

This code performs a check of the parameter password against the presidential standard of “1234” and throws an exception if there is a match, otherwise the code does nothing.

Summary

It really is that simple. There are two “flavours” of the throw method:

```
"X99: Error Msg" .throw
throw"X99: Error Msg"
```

The first form is needed when the message string needs to be constructed or contains variable information. The second form is simpler and more succinct. However, both do the same basic thing; they send an exception string to the nearest active catch clause, or the default handler in there are no active catch clauses.

43 Fortunately, this role is not nearly so perilous in fOOrth as it is when taking on the military-industrial oilagarchy.
44 This example is in the file throw.foorth in the docs/snippets folder.

fOOrth Native Exception Codes:

Exception codes generated within fOOrth all take the form “F99:” (an “F”, 2 digits, and a colon) followed by a descriptive message.

Code	Description
F	Generic fOOrth Native Exception Code for All Errors.
F1	Compile Time Errors.
F10	Syntax Error.
F11	Syntax (Specification) Error.
F12	Control Structure Nesting Error
F13	Invalid Operation for Target.
F2	Message Passing Errors
F20	Message Not Understood by the Receiver.
F21	Control Structure is Not Supported by the Receiver.
F3	Data Underflow Errors
F30	Virtual Machine Data Stack Underflow
F31	Stack/Queue Underflow
F40	Data Conversion Error
F41	Invalid loop increment value: <value>
F5	I/O Errors
F50	Error Opening a File for Reading
F51	Error Opening a File for Writing
F6	Thread Errors
F60	Duplicate Virtual Machines
F90	Internal Compiler Data Structure Error

Application Error Codes:

In general, convention indicates that application specific error codes begin with an upper case letter[45] and a two digit code. Further any optional sub-code is placed after a comma (“,”). Some possible interpretations of leading letters is shown below:

Code	Description
Ann	Generic Application Errors.
Cnn	Communication Errors.
Dnn	Database Errors.
Inn	Internal Errors.
Nnn	Network Errors.
Unn	User/Authentication Errors.
Xnn	Unknown or Unspecified Errors.

Ruby Mapped Exception Codes:

Exceptions generated by Ruby are mapped to fOOrth exceptions. There are a lot of Ruby exceptions, so it is a rather lengthy map.

Code	Description
E	Generic Ruby Mapped Exception Code for All Standard Errors.
E01	Argument Error
E01,01	Gem::Requirement::Bad Requirement Error
E02	Encoding Error
E02,01	Encoding::Compatibility Error
E02,02	Encoding::Converter Not Found Error
E02,03	Encoding::Invalid Byte Sequence Error
E02,04	Encoding::Undefined Conversion Error
E03	Fiber Error
E04	I/O Error
E04,01	EOF Error

45 It is strongly recommended that prefix codes starting in “E” and “F” be avoided.

Code	Description
E05	Index Error
E05,01	Key Error
E05,02	Stop Iteration Error
E06	Local Jump Error
E07	Math::Domain Error
E08	Name Error
E08,01	No Method Error
E09	Range Error
E09,01	Float Domain Error
E10	Regular Expression Error
E11	Runtime Error
E11,01	Gem::Exception
E11,01,01	Gem::Command Line Error
E11,01,02	Gem::Dependency Error
E11,01,03	Gem::Dependency Removal Exception
E11,01,04	Gem::Dependency Resolution Error
E11,01,05	Gem::Document Error
E11,01,06	Gem::End Of YAML Exception
E11,01,07	Gem::File Permission Error
E11,01,08	Gem::Format Exception
E11,01,09	Gem::Gem Not Found Exception
E11,01,09,01	Gem::Specific Gem Not Found Exception
E11,01,10	Gem::Gem Not In Home Exception
E11,01,11	Gem::Impossible Dependencies Error
E11,01,12	Gem::Install Error
E11,01,13	Gem::Invalid Specification Exception

Code	Description
E11,01,14	Gem::Operation Not Supported Error
E11,01,15	Gem::Remote Error
E11,01,16	Gem::Remote Installation Canceled
E11,01,17	Gem::Remote Installation Skipped
E11,01,18	Gem::Remote Source Exception
E11,01,19	Gem::Ruby Version Mismatch[46]
E11,01,20	Gem::Unsatisfiable Dependency Error
E11,01,21	Gem::Verification Error
E12	System Call Error[47]
E12,E2BIG	Argument list too long
E12,EACCES	Permission denied
E12,EADDRINUSE	Address already in use
E12,EADDRNOTAVAIL	Cannot assign requested address
E12,EAFNOSUPPORT	Address family not supported by protocol
E12,EAGAIN	Try again
E12,EAGAINWaitReadable	?
E12,EAGAINWaitWritable	?
E12,EALREADY	Operation already in progress
E12,EBADF	Bad file number
E12,EBUSY	Device or resource busy
E12,ECHILD	No child processes
E12,ECONNABORTED	Software caused connection abort
E12,ECONNREFUSED	Connection refused
E12,ECONNRESET	Connection reset by peer
E12,EDEADLK	Resource deadlock would occur
E12,EDESTADDRREQ	Destination address required
E12,EDOM	Math argument out of domain of function
E12,EDQUOT	Quota exceeded
E12,EEXIST	File exists
E12,EFAULT	Bad address

46 This exception is only supported in Ruby 2.1.x and later.

47 System Call errors are wrappers around operating system error codes. As these vary by operating system, the list that follows is typical, but by no means exhaustive or required. Refer to operating system error code documentation for more information on these errors. The descriptions here are from Linux documentation.

Code	Description
E12,EFBIG	File too large
E12,EHOSTDOWN	Host is down
E12,EHOSTUNREACH	No route to host
E12,EILSEQ	Illegal byte sequence
E12,EINPROGRESS	Operation now in progress
E12,EINPROGRESSWaitReadable	?
E12,EINPROGRESSWaitWritable	?
E12,EINTR	Interrupted system call
E12,EINVAL	Invalid argument
E12,EIO	I/O error
E12,EISCONN	Transport endpoint is already connected
E12,EISDIR	Is a directory
E12,ELOOP	Too many symbolic links encountered
E12,EMFILE	Too many open files
E12,EMLINK	Too many links
E12,EMSGSIZE	Message too long
E12,ENAMETOOLONG	File name too long
E12,ENETDOWN	Network is down
E12,ENETRESET	Network dropped connection because of reset
E12,ENETUNREACH	Network is unreachable
E12,ENFILE	File table overflow
E12,ENOBUFS	No buffer space available
E12,ENODEV	No such device
E12,ENOENT	No such file or directory
E12,ENOEXEC	Exec format error
E12,ENOLCK	No record locks available
E12,ENOMEM	Out of memory
E12,ENOPROTOOPT	Protocol not available
E12,ENOSPC	No space left on device
E12,ENOSYS	Function not implemented
E12,ENOTCONN	Transport endpoint is not connected
E12,ENOTDIR	Not a directory
E12,ENOTEMPTY	Directory not empty
E12,ENOTSOCK	Socket operation on non-socket
E12,ENOTTY	Not a typewriter... what's a typewriter?

Code	Description
E12,ENXIO	No such device or address
E12,EOPNOTSUPP	Operation not supported on transport endpoint
E12,EPERM	Operation not permitted
E12,EPFNOSUPPORT	Protocol family not supported
E12,EPIPE	Broken pipe
E12,EPROCLIM	?
E12,EPROTONOSUPPORT	Protocol not supported
E12,EPROTOTYPE	Protocol wrong type for socket
E12,ERANGE	Math result not representable
E12,EREMOTE	Object is remote
E12,EROFS	Read-only file system
E12,ESHUTDOWN	Cannot send after transport endpoint shutdown
E12,ESOCKTNOSUPPORT	Socket type not supported
E12,ESPIPE	Illegal seek
E12,ESRCH	No such process
E12,ESTALE	Stale NFS file handle
E12,ETIMEDOUT	Connection timed out
E12,ETOOMANYREFS	Too many references: cannot splice
E12,EUSERS	Too many users
E12,EWOULDBLOCK	Operation would block
E12,EWOULDBLOCKWaitReadable	?
E12,EWOULDBLOCKWaitWritable	?
E12,EXDEV	Cross-device link
E12,NOERROR	Nothing to see here. Move along, move along.
E13	Thread Error
E14	Type Error
E15	Zero Division Error
E30	Generic Ruby Mapped Exception Code for Signal Exceptions.
E30,01	Interrupt (Typically Control-C)

A Brief Overview of Key OO Concepts

A detailed study of object oriented programming principles is far beyond the scope of this User's Guide. What follows is a very brief overview of these concepts as they apply to fOOrth.

In the world of object oriented programming, there are many diverse domains of thought. In the fOOrth language system, two of those domains have their expression:

- Class based object oriented design.
- Prototype based object oriented design.

Class Based OO

Going all the way back to Smalltalk in 1980, the most common sort of object oriented systems are those that are class based. Classes act as a sort of shared hierarchical label or category for objects (commonly referred to as instances) of that class. Classes provide two main services to the programming environment:

1. They are containers for units of code called methods. These methods are available to all instances of the class as well as any classes based on the class. Reflecting the communal nature of these methods, they are called shared methods in fOOrth. Method definitions in a class override those of classes higher up in the hierarchy.
2. Classes also act as factories or templates for creating instances of their class. This is often done with the .new method applied to the class in question but there are other methods as well. Refer to documentation for the class in question in the reference below.

All object oriented behavior flows from these two simple behavioral properties.

In some languages (like C++) classes are not objects. They exist primarily as constructs used by the compiler to generate code. This is not the case with fOOrth where classes are themselves full fledged objects. Given that they are objects, this means that they too are instances of a class. In the case of classes, it turns out that they are instances of the Class class. Now the Class class is also an object and an instance of a class but what class? Perhaps some SuperClassyClass? Nope; The Class class is unique in that it is an instance of *itself*. This unusual condition is shared by all Smalltalk like languages.

Class based object oriented programming is utilized by a vast number of modern programming languages, far too numerous to mention.

Class Based Inheritance

Classes are related to one another through inheritance. A single class, called Object, is the root of the entire tree of classes. This nested hierarchy of classes in the fOOrth system is illustrated below:

```
Object
     Array
     Class
     Duration
     FalseClass
     Hash
     InStream
     Mutex
     NilClass
     Numeric
          Complex
          Float
          Integer
               Bignum
               Fixnum
          Rational
     OutStream
     Procedure
     Queue
     String
     Thread
     Time
     TrueClass
     VirtualMachine
```

Classes (other than Object) inherit behaviors from their ancestor or parent classes. Thus the capabilities of objects is able to “layer” over the capabilities of their parent classes. This property can be used for a number of different effects:

Code Reuse: Code in the parent class is effectively shared by all classes the are derived from the base class. By default the subclass retains all of the operations of the base class. Thus code common to several classes can be “factored-out” to a common base class.

Specialization: Sub-classes can be created that are more specialized variants of the base class. For example: If a class called Animal existed, a subclass of Animal called Mammal would have the characteristics of an animal plus the special characteristics of a mammal.

For a concrete example in the fOOrth hierarchy consider that the Integer class is a more specialized version of the Numeric class.

Restriction: Sub-classes can be created to limit the behavior of the class. Consider a Document class and its subclass ReadOnlyDocument. The ReadOnlyDocument would limit and restrict actions that might modify the document but would otherwise inherit other behavior from the Document base class.

In fOOrth, the Complex class restricts actions in the Numeric base class that require the value to be treated as a magnitude. Complex numbers are not magnitudes so these actions (such as the > operator) are restricted.

Prototype Based OO

While class based inheritance is the classical approach to sharing behaviors, there is another way this can be done: Prototype based object oriented design.

In this approach, there exists no special hierarchy of classes. Instead, all objects can take on the crucial two tasks normally ascribed to classes.

1. They are containers for units of code called methods. These methods are bound to individual objects. Reflecting the limited nature of these methods, they are called exclusive methods in fOOrth. Method definitions in a object override older method definitions for that object.
2. Objects can also act as factories or templates for creating instances of themselves. This is often done with the .clone method applied to the class in question but there may be other methods as well.

In practice, a prototypical object is created and its attributes are set up once. Then when more instances of this object are needed, it is simply cloned. The clones can then function as separate entities from the original. They can even have additional attributes added to then and can then serve as prototypes themselves. In this way prototype based inheritance is also supported.

In this way, the goals of *Specialization* and *Restriction* discussed above may be achieved in a prototype based system without the use of classes.

The primary language based on prototypes is ECMAScript[48] (aka JavaScript).

Methods in fOOrth

All code in fOOrth is contained in methods. A method is a fragment of code that an object uses to respond to a message that has been sent to that object. In fOOrth there are three sorts of methods:

- Shared: Methods that are common to all instances of the class that contains them.
- Exclusive: Methods that are defined for one and only one object (and all of its clones that are created *after* the exclusive method is defined.).
- Local: Methods that are created in a context and are accessible only in that context. When the context concludes, these methods are no longer accessible.

48 Even though Ruby tolerates prototype based programming, it does not really embrace it.

Late Binding and Polymorphism

Some programming languages like C++ and Java use classes and inheritance as the basis for data typing, late binding and polymorphism. This is not the case in fOOrth[49].

In fOOrth, the connection between a message and the object that is to receive that message does not occurs until the message is actually sent at run time. That is what late-binding is all about.

A side effect of this is that the receivers of messages need only be capable of responding to the specific messages sent to them. They are not required to be part of a certain class sub-tree, or have some connection to a known base class. Thus polymorphism is achieved through message interface compatibility or “duck” typing.

Summary

Given that both class and prototype based object oriented design are present in fOOrth, it is anticipated that well designed fOOrth programs will take a hybrid approach, using each system where it is best suited to the task at hand.

For example, rather than starting with a generic object and adding all needed attributes to that object, a more advanced class may be utilized and then extended to create a prototype for new objects without creating another class.

49 Or in the underlying Ruby implementation, for that matter.

Method Mapping

In fOOrth, method names are represented as simple strings. In the underlying Ruby implementation, specialized “symbol” objects are used for this task. In running fOOrth on top of an existing Ruby system, there were a number of issues that needed to be resolved.

1. The strings used by fOOrth needed to be converted to Ruby symbols to allow code to be executed in Ruby.
2. The symbols used indirectly by fOOrth can not be allowed to conflict with the myriad of symbols already in use by Ruby. Symbol collisions would cause the language system to fail catastrophically as methods were redefined in an incoherent manner.
3. The mapping of symbols had to allow some strings to map to known symbols so that Ruby code code be constructed that uses those symbols for internal actions.

In the reference implementation of fOOrth this mapping task is the responsibility of the SymbolMap class. This class creates mappings in one of two ways:

1. Be default, a symbol is generated of the form :_dddd where dddd represents a counting sequence of digits (not limited to four digits by the way). Since Ruby has no methods or symbols defined in this way, the odds of a symbol space collision are very low indeed.
2. The alternative is to explicitly specify the symbol used in the mapping. This special case is used when Ruby code needs to send or otherwise use a fOOrth symbol. In this case, the programmer is responsible for ensuring that no name space collisions occur.

Exploring the mapping system

The user is able to explore the symbol table through two command commands provided for that purpose, the)map” and)unmap” methods. These are demonstrated below:

```
>)map"+"
+ => _016

>)unmap"_016"
_016 <= +
```

In the above example, the addition (“+”) operator maps to the symbol _016. The next line shows _016 mapping back to addition. The next example shows a method name with a “custom” mapping:

```
>)map".init"
.init => foorth_init

>)unmap"foorth_init"
foorth_init <= .init
```

Another command for exploring the symbol mapping space is the)entries command. This

command generates a listing of all symbols currently defined in the system at that point in time. The command generates a paginated, formatted output.

```
>)entries
Symbol Map Entries =
!               .[]!            .getc           .reverse        0>
!:              .[]@            .gets           .right          0>=
"               .abs            .hypot          .right?         1+
… <voluminous output deleted>
```

An example of the output of the)entries command is found in the section Appendix A – Symbol Glossary.

Context

Whenever code is executed in anyway in fOOrth, whether interactively, loading a file, or compiling a method, it does so in the presence of a context. In fOOrth, the context is a nested description of the current system's conditions. When a new level of nesting is encountered, an new layer of context is added.

The context concept is useful for keeping track of important information for the compiler. Among this information is:

Tag	Description
virtual machine	The VM associated with this context.
mode	The compiler mode: execute, deferred or compile modes.
ctrl	The control tag associated with this nest structure.
cls	The class that is the target of this operation.
obj	The object that is the target of this operation.
action	A pending action to be performed when a control structure is completed.
local methods	Any local (or context) methods defined.

Note that most elements of context are optional and do not always occur.

Exploring Context

At this time, the context system is not available at the fOOrth programming level, however, a few methods are available for exploring the context system. The first is the command)context. This command lists the current context information at the console. For example:

```
>)context

Context level 1
virtual machine => VirtualMachine instance <Main>
mode => execute
```

Now, context is largely used by the compiler, so another method)context! is more useful. This method has the immediate attribute, meaning that it executes, even when the mode is deferred or compiling. This allows us to take a peek at the context inside of the working compiler. Consider these examples:

```
>true if )context! else )context! then

Context level 2
```

```
ctrl => if
mode => deferred
"else" => #<XfOOrth::LocalSpec:0x21eaf30>
"then" => #<XfOOrth::LocalSpec:0x21eaea0>

Context level 1
virtual machine => VirtualMachine instance <Main>
mode => execute

Context level 2
ctrl => if
mode => deferred
"then" => #<XfOOrth::LocalSpec:0x21eaea0>

Context level 1
virtual machine => VirtualMachine instance <Main>
mode => execute
```

The first)context! executes in the body of the “if” clause and the second one executes in the body of the “else” clause. Note how a local method “else” is defined in the first block, but is no longer defined in the second context listing. This reflects the fact that only one “else” clause is allowed in a “if” statement.

Another example is a simple method:

```
>: fred )context! dup + ;

Context level 2
mode => compile
ctrl => :
action => #<Proc:0x22702e0@C:/.../compile_library.rb:17 (lambda)>
virtual machine => VirtualMachine instance <Main>
"var:" => #<XfOOrth::LocalSpec:0x22700d0>
"val:" => #<XfOOrth::LocalSpec:0x2270010>
"var@:" => #<XfOOrth::LocalSpec:0x226bed0>
"val@:" => #<XfOOrth::LocalSpec:0x226bd98>
"super" => #<XfOOrth::LocalSpec:0x226bc00>
";" => #<XfOOrth::LocalSpec:0x226bb10>

Context level 1
virtual machine => VirtualMachine instance <Main>
mode => execute
```

The context activity reflects the activities of the compiler in compiling the code. The mode is compile mode, the ctrl is a “:” since that started the compile, an action is pending to attach the method when the compilation is completed and several local methods are defined. These include methods for local and instance data, access to the super-method, and the “;” that ends the compilation process.

The)context! method may be used to gain insight into the compilation process. Since it executes immediately, it does not affect the code generated.

Tracking the Virtual Machine

As important as the context is to the state of the compiler, it is only part of the story. The other major player in this drama is the Virtual Machine itself. So fOOrth provides two introspection methods to reveal key points in the state of the VM. These commands are)vm and its immediate version)vm!.

The following is the VM state at the console prompt:

```
>)vm

VirtualMachine instance <Main>
  Ruby    = #<XfOOrth::VirtualMachine:0x22049a0>
  Stack   = []
  Nesting = 1
  Quotes  = 0
  Debug   = false
  Show    = false
  Force   = false
  Start   = 2015-05-03 09:49:08 -0400
  Source  = The console.
  Buffer  = ")vm"
```

The snippets file show.foorth reveals the tracking of code sources when loading code:

```
>)load"docs/snippets/show
Loading file: docs/snippets/show.foorth

VirtualMachine instance <Main>
  Ruby    = #<XfOOrth::VirtualMachine:0x22049a0>
  Stack   = []
  Nesting = 1
  Quotes  = 0
  Debug   = false
  Show    = false
  Force   = false
  Start   = 2015-05-03 09:49:08 -0400
  Source  = A file: docs/snippets/show.foorth
  Buffer  = ")vm"
Completed in 0.1 seconds
```

As can be seen, the VM has its vital stack, source, and tracking for quotes and structure nesting as well as various optional modes.

Note that the Force flag is a hold-over from FORTH. This flag overrides the immediate status of the next word compiled, forcing it to be compiled rather than executed. At this point, this feature is not employed in fOOrth. Like the return stack, this may be removed at some point if it does not prove useful.

Routing

Message routing in fOOrth has two separate but interrelated aspects: When methods are being defined a method specification must be created and stored in the appropriate location. This creates routing information. Then when methods are being compiled into code, the compiler must be able to locate the correct method specification so that the correct output code is generated. This uses routing information.

The key responsibility of these specifications is to determine the message receiver when the method is compiled. This is one area where the terse, concise RPN of fOOrth can be a liability. The lack of redundancy sometimes makes it difficult to determine the intended message receiver.

In FORTH, this is never an issue since all words exist as subroutines of the virtual machine. In fOOrth, a number of factors determine the type of method and thus its routing. These are:

1. The defining word used to create the method.
2. The receiver of the defining word used to create the method.
3. The name of the method.

The various method mapping and routing targets are reflected in the ways by which methods may be defined. These next sections review the types of methods, their mapping and routing, and how they are used in the fOOrth language system.

Virtual Machine Methods

Virtual Machine methods are the part of fOOrth that comes closest to classical FORTH. In these methods the target of the method is the current virtual machine associated with the executing thread and the compiler that created the method in the first place. Since there is always a virtual machine present, these methods are never out-of-context. This allows these methods alone to support immediate mode in which methods are executed even when the system is in a deferred or compile state.

To define a standard Virtual Machine method use:

```
: name (method body here) ;
```

To define an immediate method use this similar format:

```
!: name (method body here) ;
```

The rules for naming virtual machine methods are the most liberal. They must not contain spaces, and if a " is present it is the last character in the name and a string literal is part of the method name when run. See String Literals in the section the Syntax and Style of fOOrth.

Shared Methods

Shared methods are those methods that are shared by all the member of a class and its sub-classes. All of the different sorts of shared methods a defined using the following construction:

```
A_Class .: method_name (method body here) ;
```

The method name may not contain spaces. Further the method name is responsible for determining the routing used and any embedded string data.

The lead character determines the routing:

- “.” - indicates that the message is routed to the top-of-stack element (TOS)
- “~” - indicates that the message is routed to the “self” entity of the method (see the section Self below). Since these messages are only routed to the object itself, they are in effect private methods.
- Other – these methods are used to implement dyadic operators. To do this they are routed to the next element on the stack (NOS). This corresponds to the “left binding” of dyadic operators.

The presence of a " character in the method name indicates a trailing string is embedded in the method name. See String Literals in the section the Syntax and Style of fOOrth. Shared methods add an additional criteria on the use of embedded strings:

If the shared method is routed to TOS and there is an embedded string, then the class of the method must be String, otherwise an error is reported.

Exclusive Methods

Exclusive methods are defined on an individual object as opposed to all the instances of a class of objects. All of the different sorts of shared methods are defined using the following construction:

```
an_object .:: method_name (method body here) ;
```

The method name may not contain spaces. Further the method name is responsible for determining the routing used and any embedded string data.

The lead character determines the routing:

- “.” - indicates that the message is routed to the top-of-stack element (TOS)
- “~” - indicates that the message is routed to the “self” entity of the method (see the section Self below). Since these messages are only routed to the object itself, they are in effect private methods.
- Other – these methods are used to implement dyadic operators. To do this they are routed to the next element on the stack (NOS). This corresponds to the “left binding” of dyadic operators.

The presence of a " character in the method name indicates a trailing string is embedded in the method name. See String Literals in the section the Syntax and Style of fOOrth. Shared

methods add an additional criteria on the use of embedded strings:

If the shared method is routed to TOS and there is an embedded string, then the class of the method must be String, otherwise an error is reported.

Shared Stub Methods

– Not implemented. Currently part of the implementation but not yet part of the language.

Exclusive Stub Methods

– Not implemented. Currently part of the implementation but not yet part of the language.

Local Methods

– Not implemented. Currently part of the implementation but not yet part of the language.

The various combinations of the above are summarized below:

Defining Word	DW Receiver	Method Name	Message Routing	Notes
:	N/A	`any`[50]	VM[51]	A virtual machine method.
.:	A Class	`.name`	TOS[52]	A public shared instance method.
	A Class	`~name`	Self[53]	A private shared instance method.
	A Class	`other`	NOS[54]	A public shared dyadic operator.
	N/A	`invalid`[55]	N/A	Not allowed: Error
.::	A Class	`.name`	TOS	A public class method
	An Object	`.name`	TOS	An public exclusive instance method.
	A Class	`~name`	Self	A private class method
	An Object	`~name`	Self	A private exclusive instance method.
	A Class	`other`	NOS	A public class exclusive dyadic operator
	An Object	`other`	NOS	A public exclusive dyadic operator
	N/A	`invalid`	N/A	Not allowed: Error
tbd[56]	N/A	`other`	Context	A context local method.

50 The names of Virtual Machine methods have no restrictions except that they contain no spaces.
51 The message receiver is the Virtual Machine.
52 The message receiver is the Top element of the Data Stack.
53 The message receiver is the implicit “self” of the method owner.
54 The message receiver is the Second element of the Data Stack.
55 Any method beginning with an upper case letter or $ or # or @ is invalid and will generate an error.
56 At this time, local methods are cannot be defined by the programmer. They are all builtin methods.

Routing Internals

As code is being compiled, the specification for each method must be located. This task is performed by the virtual machine's linked list of context objects. This is the sensible place for this to occur as routing is context sensitive.

The process of routing involves searching for the specification in an ordered list of places. This list is sensitive to the name of the method being processed as well as the target of the compilation process and any local, context sensitive definitions. This is summarized below:

Method	**.name**	**~name**	**@name**	**$name**	**#name**	**other**
Filter Regex	/^\./	/^~/	/^@/	/^\$/	/^#/	Other
Search List	Object VM TOS	Target Class Target Object VM Self	Target Class Target Object VM Error	Global Error	VM Error	Local Object VM Global Error

Where the search list entries are defined as:

Search Location	**Search Description**
Object	Search the class Object for a specification.
VM	Search the class VirtualMachine for a specification.
Target Class	Search the class targeted by this compile action (if any) for a specification. Note this also searches any parent classes of the target class.
Target Object	Search the object targeted by this compile action (if any) for a specification. Note this also searches the object's class and any parent classes of the that class.
Local	Search the compiler context tree for a specification.
Global	Search the global name space for a specification.
TOS	Assume that this method uses TOS routing and create a temporary default specification.
Self	Assume that this method uses Self routing and create a temporary default specification.
Error	Unable to find a specification. Signal an error. (See Spec Error below)

Spec Errors

When compiling a token into a method or looking it up to executed interactively, the fOOrth virtual machine performs several checks based on the text of the language token. A Spec Error is reported when this search is unable to locate a specification for the token. A Spec Error takes the form:

```
F11: ?name?
```

There are two basic ways that a spec error may be encountered:

Method is Out of Context

The first of these is that the programmer has written code that contains context sensitive methods, outside of that context. As a classic case of this consider the “if” and the “else” methods. Normally “else” only makes sense if there is an “if” for it to be “bound” to. Thus if you simply enter “else” there are potentially two outcomes:

```
>else

F10: ?else?
```

Or:

```
>else

F11: ?else?
```

In the first case, the system has never encountered an “if” statement, so “else” is completely undefined. In the second case , “if” statements have been encountered, but the “else” is out of context. Given how common “if” statements are, this second error message is far more likely, but for rarer control structures, the first error may be seen.

Incorrect Routing Information

The second way to get a spec error is caused by the routing information not being set up correctly. To be clear, this should not happen. They are a sort of internal compiler error that things are not quite right. As such this version needs to be reported as a bug.

As this error should not happen, there is currently no example of this case. However, should one occur, it would likely take the form of a spec error on a method that *should* be in the correct context.

Self

Whenever code executes in the fOOrth language system, there is an object that “owns” that code. Even code run at the console interactively belongs to an object, the virtual machine.

In fOOrth, the “self” method gives the programmer explicit access to this owning object. Let's try this from the console:

```
>self .
VirtualMachine instance <Main>
```

The console code is run in the context of an instance of a Virtual Machine named “Main”. What about some methods? Here is a simple method:

```
>: show_self self . ;

>show_self
VirtualMachine instance <Main>
```

This has the same owner which is not unexpected as show_self is a virtual machine method. How about something more interesting:

```
>class: MyClass

>MyClass .: .show_self self .name . ;

>MyClass .new .show_self
MyClass instance
```

For this shared method on the class MyClass, self is an instance of MyClass.

Applying Self

So far, the self value seems pretty academic. Let's see the ways this value affects fOOrth code:

1. The self is the target for self routed methods. These are methods that begin with “~” (see Shared Methods and Exclusive Methods above). The use of self routing has big savings. There is no need to retrieve the value, “~” methods can act on it without any preamble. There is no need to worry about where the receiver is on the stack, self is always available.
2. For TOS routed methods, access to self is as simple as “self .method_name”. This was done in the MyClass example above. The self value is a free value that does not to be declared.
3. When instance values and variables (see Data Storage in fOOrth above) are created, they are always applied to the self object. Access to these values and variables is also in reference to self. Thus, by default, these data are only directly accessible inside methods of the object holding those data.

4. In some methods that take a block, the self in the block is often very useful. A good example of this is in file I/O classes line InStream (see below). The InStream .open{ … } method for example. In the block (delimited by the { and } characters) self is defined to be the InStream instance. This allows easy access to the file data with "~" methods.

Changing Self

There are times that we all wished we were someone else. In a fOOrth program there are instances where it would be advantageous to have self be something else. This is accomplished using the .with{{ … }} construction. A few simple examples:

```
>"Test String" .with{{ self .name . }}
String instance
>MyClass .new .with{{ self .name . }}
MyClass instance
```

The receiver of the ".with{{" method becomes the "self" within the block it defines. The uses of these control structures include those listed above, but there are some additional points specific to this application:

1. By applying a .with{{ … }} clause, the program gains access to "~" methods of the target. In addition, it also allows access to instance values and variables of that object. On the downside, it removes access to "~" methods and instance values and variables of the method the .with{{ … }} clause was contained in.
2. This clause makes it easy to define new instance values and variables to an individual object. This is much easier to do than creating an exclusive method and then executing it once. This facilitates the setup of prototype objects (see Prototypes above).
3. A .with{{ … }} clause can be used to take a value and give easy access to that value within the executed block without having to create a local value.
4. Access to the self value is generally faster than other forms of access. Thus this construct can yield performance enhancements.

Boolean Data

In fOOrth, there is no class called Boolean. Instead, the functionality of Boolean logic are built into other classes, and not always the class you'd expect. Thus Boolean data could use a little explanation and clarification.

What values represent true and false?

This is fundamental to the operation of fOOrth and in this area, fOOrth pretty much follows Ruby's lead. Here is a complete overview of what constitutes true and false:

False Values	True Values
false nil	Everything else!

That's it. In particular, the number 0, and the empty string are true, and *not* false.

Processing Boolean Data

In processing Boolean data, fOOrth takes the common approach of processing from the perspective of the true or false message receiver. This technique goes back to the very earliest object oriented programming systems. What may surprise is where this processing takes place. This is illustrated below:

False Processing	True Processing
FalseClass NilClass	Object

Note that while false processing occurs in FalseClass and NilClass, true processing occurs in Object and not TrueClass. In fact the only purpose for TrueClass is to be the class for the value true. It has no methods of its own. All of the work processing true values is in the Object class.

Boolean Constants

Boolean value constants are: true, false and nil. These values may be used in any context.

Numeric Data

The fOOrth language system has an elaborate level of support for numeric data that bears close examination in its own right. The numeric class tree consists of these classes:

Class	Description
Numeric	The abstract base class for all concrete numeric values.
Complex	Complex numbers in the form a+bi where “a” and “b” represent real numbers and “i” represents the square root of -1.
Float	The class of floating point numbers based on the IEEE standard.
Integer	An abstract class for whole numbers
Bixnum[57]	The class of really really big whole numbers
Fixnum	The class of whole number that fit into one memory word.
Rational	Rational number in the form a/b where “a” and “b” are whole number and “b” is not zero.

In this system most operations are defined at the level of numeric with a few exceptions:

- Integer defines a few “bit” oriented operations that are specific to whole numbers.
- Complex stubs out several methods that require values to be comparable as magnitudes. Complex numbers are not magnitudes so these operations are invalid. See the Complex class below for more details on the methods that are affected.

57 The presence of the Bignum and FixNum classes is a case of the Ruby implementation “leaking” through to the fOOrth language. For operations on integer values, the use of the Integer class is strongly recommended.

Procedure Data

It may be odd to consider that code (procedures) be treated as data, but in fOOrth, this is exactly the case. With the Procedure class of objects, procedures can be created, stored, and passed as arguments to methods.

In addition, fOOrth supports Procedure Literal values, in much the same way that it supports numeric or string literal values. With strings, any method whose name ends with a double quote mark (") will be followed by a string literal that ends when the matching closing double quote mark is encountered.

It is similar with procedures. Any method whose name ends with “{{“ contains a procedure literal value that ends when the matching “}}” is found. The simplest example of this is the “{{“ with no method name:

```
{{ dup + }}
```

This code creates a simple procedure and leaves a reference to that procedure on the stack. This is similar to the action of the string literal:

```
"Testing 1 2 3"
```

Which leaves a reference to the string “Test 1 2 3” on the stack.

Values and Indexes

A very common use for procedure literals is to serve as the “brains” for action oriented methods like .each{{. As can be seen, the .each{{ method ends with {{ so a procedure literal follows. Like many such methods, the .each{{ method needs to send some additional data to the procedure. These data are a value and the index associated with that value. Inside the procedure these are accessed with the local methods “v” and “x” respectively.

Note: If no value or index are defined in the current context, then the local methods “v” and “x” simply return the value nil.

A fOOrth Reference

The following sections contain a class by class reference to the fOOrth language. For each class, a number of sub-sections, some optional, are present. These are:

- A summary of the class's inheritance. For example: “Inheritance: Array ← Object”
- A summary of the various sorts of class methods, instance methods, stubs, and helper methods. Some or all of these may be absent if they are not present in the class under discussion.
- A description of literal values of this class, if they are supported.
- A description of other optional attributes of the class such as special methods for creating instances of the class, special values, formatting, parsing, etc.
- Class methods. These are methods that bind to the class object itself. That is, class methods are exclusive methods of the class object.
- Instance methods. These are methods that bind to instances of the class being discussed. That is, class methods are shared methods of the class object.
- Stub methods. These are methods that are disallowed for this class, or placeholders for classes derived from this class. They can be either class methods or instance methods. Attempting to send a stub method will result in an error.

A typical method looks description like:

[array object] + [array]

Routing: NOS

This method overrides the stub defined in the Object class. For arrays, the plus operation is defined as concatenation. The result is a new array with the object appended. If the object is also an array, the elements of that array are concatenated.

Note: This method does *not* mutate the original array.

Code	Result
`[ 1 2 ] 3 +`	[1 2 3]
`[ 1 2 ] [ 3 4 ] +`	[1 2 3 4]
`[ 1 2 ] [ [ 3 4 ] ] +`	[1 2 [3 4]]

Where the title line is a summary of the action of this method. The first [ ] describes the required conditions on the stack before the method is executed, the method name along with possible

embedded arguments follows, and the trailing [ ] describes conditions on the stack after the method has executed.

The routing line describes the type of message routing used. These can be VM, TOS, NOS, Self, or Compiler Context. See Routing above for more details.

Then a (clear, concise and illuminating) description of the method and any noteworthy or cautionary information follows.

Finally, a series of examples, depicting some sample code and the results (on the stack) of executing that code.

Note that some methods have further sections that describe any local methods created within its context. These are described under the sub-heading of “Local Methods”. Local methods only exist within the context of the methods that create them. When that context ends, those methods are no longer accessible.

Array

Inheritance: Array ← Object

```
Array Class Methods =
.new_size   .new_value  .new_values .new{{

Array Shared Methods =
!             .-midlr       .map{{        .select{{     .to_t
+             .-right       .max          .shuffle      .to_t!
.+left        .[]!          .mid          .sort         <<
.+mid         .[]@          .midlr        .split        @
.+midlr       .each{{       .min          .strmax
.+right       .empty?       .pp           .to_duration
.-left        .left         .reverse      .to_duration!
.-mid         .length       .right        .to_s

Helper Methods =
.join     .new      [
```

Array objects[58] are collections of data indexed by an integer value from 0 through N where N is an arbitrary, non-negative, non-stellar, whole number. The fOOrth language system supports the creation of array literals and has several methods for putting data into and pulling data out of arrays.

Array Literals

Array literals are supported by the virtual machine method "`[`" and a locally defined method "`]`". The general usage is:

```
[ (data generating code goes here) ]
```

Where the data generation code is code that deposits zero or more data elements onto the stack. When the closing "`]`" is encountered, these data elements are scooped up and placed into an array at the top of the stack. Here are some illustrations of array literals in action:

```
>[ ] .
[ ]
>[ (data generating code goes here) ] .
[ ]
>[ 1 2 3 ] .
[1, 2, 3]
>[ 2 "for" 1 true ] .
[ 2 "for" 1 true ]
```

Some points of interest:

- The first two examples both create "empty" arrays with zero data elements.

58 Please see http://en.wikipedia.org/wiki/Array_data_structure for more information.

- Array data do NOT need to be the same “type” of data. Mixing is allowed.
- Any statements that generate data are permitted. Consider:

```
>[ 1 11 do i loop ] .
[ 1 2 3 4 5 6 7 8 9 10 ]
>[ 1 11 do i dup * loop ] .
[ 1 4 9 16 25 36 49 64 81 100 ]
```

- Array literals may be nested. Just be sure to properly nest the brackets.

```
>[ 1 2 [ 3 ] ] .
[ 1 2 [ 3 ] ]
```

Array Literal Methods

[stuff] [[[stuff]]

Routing: VM

This method starts the creation of an array literal. It does so by taking the entire contents of the data stack and placing it into a holding array. This frees up the stack for the task of creating the array.

Code	Result
1 2 [	[1 2]

Local Methods:

[[stuff] d1 d2 … dn]] [stuff, [d1, d2, … dn]]

Routing: Compiler Context.

This method takes the data that has been gathered onto the stack and creates the array literal while also restoring the deeper levels of the stack.

Code	Result
1 2 [3 4]	1 2 [3 4]

Array Literals in Action

The following shows the action of the code 1 2 [3 4 5] with)show and)debug active:

```
>1 2
Tags=[:numeric] Code="vm.push(1); "
Tags=[:numeric] Code="vm.push(2); "

[ 1 2 ]
>[
Tags=[:immediate] Code="vm._214(vm); "
```

```
  nest_context
  Code="vm.squash; "

[ [ 1 2 ] ]
>>3 4 5
Tags=[:numeric] Code="vm.push(3); "
Tags=[:numeric] Code="vm.push(4); "
Tags=[:numeric] Code="vm.push(5); "

[ [ 1 2 ] 3 4 5 ]
>>]
Tags=[:immediate] Code="vm.context[:_311].does.call(vm); "
  unnest_context
  Code="vm.unsquash; "

[ 1 2 [ 3 4 5 ] ]
```

Class Methods

[Array] .new [[]]

Routing: TOS

This method is actually the default implementation inherited from the Object class. It creates a new, array object with zero data elements. It is equivalent to the array literal "`[ ]`".

Code	Result
`Array .new`	[ ]
`[ ]`	[ ]

[size Array] .new{{ … }} [[d_1, d_2, d_3 … d_{size}]]

Routing: NOS (since the Procedure Literal is TOS)

It creates an array of size elements where the values are created by the embedded procedure literal block. If no value is left on the stack, an error occurs. The current element index is available in the block via the local method "`x`". For more information on the methods local to the embedded procedure, see the Procedure class.

Code	Result
`10 Array .new{{ x }}`	[0 1 2 3 4 5 6 7 8 9]
`0 Array .new{{ x dup * }}`	[0 1 4 9 16 25 36 49 64 81]
`10 Array .new{{  }}`	F30: Data Stack Underflow: pop

[size Array] .new_size [[0_1, 0_2, 0_3 … 0_{size}]]

Routing: TOS

This method creates an array of the specified size, pre-filled with the value zero.

Code	**Result**
`5 Array .new_size`	[0, 0, 0, 0, 0]

[value Array] .new_value [[value]]

Routing: TOS

This method creates an array with a single element, value. It is equivalent to the expression “[value]”.

Code	**Result**
`42 Array .new_value`	[42]

[value size] .new_values [[$value_1$, $value_2$, $value_3$ … $value_{size}$]]

Routing: TOS

This method creates an array of the specified size and pre-filled with the specified value.

Note: If the value used is mutable, be warned that the same value is used for *all* of the array elements and a change to one element will affect *all* of them.

Code	**Result**
`false 5 Array .new_values`	[false, false, false, false, false]
`"Hello" 3 Array .new_values`	[“Hello”, “Hello”, Hello”]

Instance Methods

[value array] ! []

Routing: TOS

This method overrides the stub method defined in Object. The store data operator is normally used to store a new value via a data reference. It happens to also work with arrays, but this is an edge case. When applied to an array, the value is stored in the data element indexed by the value zero (0).

Note: This also happens to be the implementation of the "!" operator used with references in general. In that context, this method updates the value associated with a reference.

Code	Result
`7 some_array !`	(some_array[0] equals 7)
`42 myvar !`	(Updates the value of myvar to 42)

[array object] + [array]

Routing: NOS

This method overrides the stub defined in the Object class. For arrays, the plus operation is defined as concatenation. The result is a new array with the object appended. If the object is also an array, the elements of that array are concatenated.

Note: This method does *not* mutate the original array.

Code	Result
`[ 1 2 ] 3 +`	[1 2 3]
`[ 1 2 ] [ 3 4 ] +`	[1 2 3 4]
`[ 1 2 ] [ [ 3 4 ] ] +`	[1 2 [3 4]]

[width source_array target_array] .+left [array]

Routing: TOS

This method removes the first (leftmost) width elements from a copy of the target array and replaces them with the elements from the source array.

Note: This method does *not* mutate the original array.

Code	Result
`2 [ 9 ] [ 1 2 3 ] .+left`	[9 3]

[posn width source_array target_array] .+mid [array]

Routing: TOS

This method removes width elements from the target array starting at position “posn”. It then inserts the elements from the source array into the “gap”, creating a new array in the process.

Note: This method does *not* mutate the original array.

Code	Result
`1 2 [ 5 ] [ 1 2 3 4 ] .+mid`	[1 5 4]

[left right source_array target_array] .+midlr [array]

Routing: TOS

This method removes elements from the target array starting at position “left” and ending at position “right” counting from the end of the array. It then inserts the elements from the source array into the “gap”, creating a new array in the process.

Note: This method does *not* mutate the original array.

Code	Result
`1 1 [ 8 9 ] [ 1 2 3 4 5 ] .+midlr`	[1 8 9 5]

[width source_array target_array] .+right [array]

Routing: TOS

This method removes the last (rightmost) width elements from the target array and replaces them with the elements of the source array.

Note: This method does *not* mutate the original array.

Code	Result
`3 [ 8 9 ] [ 1 2 3 4 5 ] .+right`	[1 2 8 9]

[width array] .-left [array]

Routing: TOS

This method removes the first (leftmost) width elements from a copy of the source array.

Note: This method does *not* mutate the original array.

Code	Result
`3 [ 1 2 3 4 5 ] .-left`	[4 5]

[posn width array] .-mid [array]

Routing: TOS

This method removes width elements from a copy of the array starting at position “posn”.

Note: This method does *not* mutate the original array.

Code	Result
`1 2 [ 1 2 3 4 5 ] .-mid`	[1 4 5]

[left right array] .-midlr [array]

Routing: TOS

This method removes elements from the target array starting at position “left” and ending at position “right” counting from the end of the array.

Note: This method does *not* mutate the original array.

Code	Result
`1 1 [ 1 2 3 4 5 6 ] .-midlr`	[1 6]

[width array] .-right [array]

Routing: TOS

This method removes the last (rightmost) width elements from a copy of the array.

Note: This method does *not* mutate the original array.

Code	Result
`2 [ 1 2 3 4 5 6 ] .-right`	[1 2 3 4]

[value index array] .[]! []

Routing: TOS

Store the specified value at the index of the array.

Notes:

- If the index is beyond the end of the array, the array is extended to encompass the new index. Cells between the previous last element and the new one are set to nil.
- Negative indexes access elements counting from the end of the array with -1 being the last element of the array.
- This method *does* mutate the array.

Code	Result
`"Hello" 5 $myarray .[]!`	(“Hello” is stored at location 5 of myarray.)

`[ 1 2 3 ] val$: $t`
`5 5 $t .[]!`
`$t`
	[1 2 3 nil nil 5]

[index array] .[]@ [value]

Routing: TOS

Retrieve the value stored in the array at the specified index.

Notes:

- If the index does not correspond to a location within the array, the value nil is returned instead.
- Negative indexes access elements counting from the end of the array with -1 being the last element of the array.

Code	Result
`1 [ 1 2 3 4 ] .[]@`	2
`11 [ 1 2 3 4 ] .[]@`	nil
`-1 [ 1 2 3 4 ] .[]@`	4

[an_array] .each{{ … }} [unspecified]

Routing: NOS (since the Procedure Literal is TOS)

This method is the array item iterator. It processes each element of the array in turn, calling the embedded procedure literal block with the value (v) and index (x) of the current array item.

For more information on the methods local to the embedded procedure, see the Procedure class.

Code	Result
`[ "1" "2" "3" "4" ]` `.each{ v x 1+ * . space }`	(Prints out:) 1 22 333 4444
`[ "1" "2" "3" "4" ] .each{ v . space }`	(Prints out:) 1 2 3 4
`[ "1" "2" "3" "4" ] .each{ x . space }`	(Prints out:) 0 1 2 3

[an_array] .empty? [a_boolean]

Routing: TOS

Is the array argument devoid of data?

Code	Result
`[ ] .empty?`	true
`[ 1 2 3 ] .empty?`	false

[a_1 a2 … a_N N] .join [after]

Routing: TOS

Join the top N stack elements into an array. If N is negative or there are fewer than N elements available, an error occurs.

This is a helper method of the Integer class.

Code	Result
`1 2 3 4 4 .join`	[1 2 3 4]
`1 2 3 4 -4 .join`	F30: Invalid array size: .join
`1 2 3 4 44 .join`	F30: Data Stack Underflow: popm

[width array] .left [array]

Routing: TOS

This method returns an array containing the first (leftmost) width elements of the given array.

Note: This method does *not* mutate the original array.

Code	Result
`3 [ 1 2 3 4 5 6 7 ] .left`	[1 2 3]

[array] .length [count]

Routing: TOS

This method computes the number of elements contained in the given array.

Code	Result
`[ 1 2 3 4 ] .length`	4
`[ ] .length`	0

[an_array] .map{{ … }} [an_array]

Routing: NOS (since the Procedure Literal is TOS)

Construct a new array, applying the transformation block to each element. The .map method processes each element of the array in turn, calling the embedded procedure literal block with the value (v) and index (x) of the current array item The value returned by the block is used to populate the new array. If no value is returned, an error occurs.

For more information on the methods local to the embedded procedure, see the Procedure class.

Note: This method does *not* mutate the original array.

Code	Result
`[ "1" "2" "3" "4" ] .map{ v x 1+ * }`	[“1” “22” “333” “4444”]
`[ 1 2 3 4 ]` `.map{ v .odd? if v else 0 then }`	[1 0 3 0]
`[ 1 2 3 4 ]` `.map{ v .odd? if v then }`	F30: Data Stack Underflow: pop
`[ "1" "2" "3" "4" ] .map{ v 2 * }`	[“11” “22” “33” “44”]
`[ "1" "2" "3" "4" ] .map{ x 1+ 2* }`	[2 4 6 8]

[array] .max [value]

Routing: TOS

This method scans through the array searching for the element with the largest value. The type of the result will match the type of the first element of the array. If comparison with other values is not supported, an error is raised.

Code	Result
`[ 1 6 2 3 4 5 ] .max`	6
`[ "1" 6 2 3 4 5 ] .max`	“6”
`[ 1 2 3 4 "apple" ] .max`	F40: Cannot coerce a String instance to an Integer instance

[posn width array] .mid [array]

Routing: TOS

This method extracts width elements from a copy of the array starting at position “posn”. If more elements are requested than exist in the array, only available elements are returned. If the start “posn” is not a valid element index, then the value nil is returned.

Note: This method does *not* mutate the original array.

Code	Result
`2 2 [ 1 2 3 4 5 6 ] .mid`	[3 4]
`2 9 [ 1 2 3 4 5 6 ] .mid`	[3 4 5 6]
`9 2 [ 1 2 3 4 5 6 ] .mid`	nil

[left right array] .midlr [array]

Routing: TOS

This method extracts elements from the target array starting at position “left” and ending at position “right” counting from the end of the array. If the left and right are such that no elements are included, an empty array is returned. If the indexes are outside of the array, nil is returned.

Note: This method does *not* mutate the original array.

Code	Result
`1 1 [ 1 2 3 4 ] .midlr`	[2 3]
`3 3 [ 1 2 3 4 ] .midlr`	[]
`8 8 [ 1 2 3 4 ] .midlr`	nil

[array] .min [value]

Routing: TOS

This method scans through the array searching for the element with the smallest value. The type of the result will match the type of the first element of the array. If comparison with other values is not supported, an error is raised.

Code	Result
`[ 1 6 2 3 4 5 ] .min`	1
`[ "1" 6 2 3 4 5 ] .min`	“1”
`[ 1 2 3 4 "apple" ] .min`	F40: Cannot coerce a String instance to an Integer instance

[array] .pp []

Routing: TOS

This method is a pretty printer for arrays. The data in the array is displayed with in columns for an 80 character wide display and a blank line every 50 lines. The primary use of this method was in preparing the lists of method names used in this guide. No value is returned.

Code	Result
`[ 1 2 3 4 5 ] .pp`	Displays “1 2 3 4 5”

[array] .reverse [array]

Routing: TOS

This method creates a copy of the array with the elements reversed.

Note: This method does *not* mutate the original array.

Code	Result
`[ 1 2 3 4 ] .reverse`	[4 3 2 1]

[width array] .right [array]

Routing: TOS

This method extracts the last (rightmost) width elements from a copy of the array.

Note: This method does *not* mutate the original array.

Code	Result
`2 [ 1 2 3 4 ] .right`	[3 4]

[an_array] .select{{ … }} [an_array]

Routing: NOS (since the Procedure Literal is TOS)

This method is used to select elements from an array and place them in a new array. If the embedded procedure literal block of the select returns true, the element is copied. If it returns false, the element is omitted. If no value is returned, an error occurs.

For more information on the methods local to the embedded procedure, see the Procedure class.

Note: This method does *not* mutate the original array.

Code	Result
`[ 1 2 3 4 ] .select{ v even? }`	[2 4]
`[ 1 2 3 4 ] .select{  }`	F30: Data Stack Underflow: pop
`[ 1 2 3 4 ] .select{ v .odd? }`	[1 3]
`[ 1 2 3 4 ] .select{ x .odd? }`	[2 4]

[array] .shuffle [array]

Routing: TOS

This method creates a new array with the elements of the source array shuffled.

Note: This method does *not* mutate the original array.

Code	Result
`[ 1 2 3 4 5 6 7 8 9 ] .shuffle`	[3 1 5 7 4 8 6 9 2] (Typical result)

[an_array] .split [the_array_elements]

Routing: TOS

This method splits out the array, placing its elements onto the data stack.

Code	Result
`1 2 [ 3 4 ] .split`	1 2 3 4

[array] .sort [array]

Routing: TOS

Given an array, return a sorted copy of that array. Note that the elements of the array must be comparison compatible or an error is returned.

Note: This method does *not* mutate the original array.

Code	Result
`[ 4 1 5 3 6 0 ] .sort`	[0 1 3 4 5 6]
`[ 4 1 5 3 "6" 0 ] .sort`	[0 1 3 4 5 “6”]
`[ "5" 5 nil 4 ] .sort`	F12: A NilClass instance does not support <=>.
`[ 1 4 nil 7 ] .sort`	F40: Cannot coerce a NilClass instance to an Integer instance

[array] .strmax [width]

Routing: TOS

Given an array, this method determines the width of the largest string representation of an element. This method is a helper method for the .pp pretty print method.

Note: This method does *not* mutate the original array.

Code	Result
`[ 1 100 3 44 ] .strmax`	3

[an_array] .to_duration [a_duration]

Routing: TOS

A helper method for the Duration class. See that class for more details.

[an_array] .to_duration! [a_duration]

Routing: TOS

A helper method for the Duration class. See that class for more details.

[an_array] .to_s [a_string]

Routing: TOS

Convert the array to a string representation of that array.

Code	Result
`[ 1 2 3 ] .to_s`	“[1 2 3]”

[an_array] .to_t [a_time]

Routing: TOS

Convert the array to a time object. This is a helper method for the Time class.

[an_array] .to_t! [a_time]

Routing: TOS

Convert the array to a time object. This is a helper method for the Time class.

[array object] << [array]

Routing: NOS

This method appends the object to the array. If the object is an array, the array and not the elements are appended.

NOTE: This method DOES mutate the source array.

Code	Result
`[ 1 2 3 ] 4 <<`	[1 2 3 4]
`[ 1 2 3 ] [ 4 5 ] <<`	[1 2 3 [4 5]]

[array] @ [value]

Routing: TOS

This method overrides the stub method defined in Object. The fetch data operator is normally used to fetch a value from a data reference. It happens to also work with arrays, but this is an edge case. When applied to an array, the value is fetched from the data element indexed by the value zero (0).

Note: This also happens to be the implementation of the “@” operator used with references in general. In that context, this method retrieves the value associated with a reference.

Code	Result
`[ 1 2 3 4 ] @`	1
`myvar @`	an_object

Class

Inheritance: Class ← Object

```
Class Shared Methods =
)methods      .is_class?    .parent_class .to_s
)stubs        .new          .subclass:

Helper Methods =
.:            .class        .is_class?    :class
```

For all classes in fOOrth, shared methods defined on a class are expressed through instances of those classes. However, the Class class is the class of all classes. That is to say, all classes are instances of the class Class. The Class class is unique in that it is an instance of itself![59] A consequence of this is that shared methods of the Class class are class methods of all classes including the Class class.

The shared methods of the Class class mostly deal with the creation of new classes, and populating those classes with the methods and data needed to accomplish useful work.

Instance Methods

[a_class] .: method_name … ; []

Routing: VM

This method is used to define new methods for instances of the specified class as well as instances of any of its sub-classes. These methods execute with “self” set to the instance that received the message.

The text of the method name determines the type of method being created. The following rules apply to the first character of the name: A “.” indicates a public method, a “~” indicates a private method, “A” through “Z”, “@”, “$”, or “#” are not allowed. All others indicate a dyadic operator with NOS routing.

See the section Routing above for more details.

59 All in all, a real class act! See http://en.wikipedia.org/wiki/Class_(computer_programming)

Code	Result
`Object .: .one 1 ;`	(Creates a public method .one)
`Object .: ~two 2 ;`	(Creates a private method ~two)
`MyClass .: + (omitted) ;`	(Creates a dyadic (NOS) method +)
`Object .: BAD  ;`	F10: Invalid name for a method: BAD
`String .: twaddle" (stuff) ;`	(Creates a method with an embedded string.
`55 .: foobar (stuff) ;`	F13: The target of .: must be a class
`Object .: twiddle" (stuff) ;`	F13: Creating a string method twiddle" on a Object

Local Methods:

[undefined] super [undefined]

Routing: Compiler Context.

Invoke the definition of this method in the nearest parent class of this class that defines a method of the same name. The arguments to the super method must match those of the parent class's implementation of this method. The return values will also be those of the parent class. If no parent class defines a method of the same name as this one, an error is generated.

Code	Result

`class: MyClass`
`MyClass .: .name "Hi from " super + ;`
`MyClass .new .name .`
	Hi from MyClass instance

`class TestClass`
`TestClass .: .broken super ;`
`TestClass .new .broken`
	F20: A TestClass instance does not understand .broken (:_309).

[value] val: local_name []

Routing: Compiler Context.

This method defines a local value in the current method.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 val: limit`	(Creates a value named limit set to 10)

[value] var: local_name []

Routing: Compiler Context.

This method defines a local variable in the current method.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 var: limit`	(Creates a variable named limit set to 10)

[value] val@: @instance_name []

Routing: Compiler Context.

This method creates an instance value in the instance of the host class.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 val@: @limit`	(Creates an instance value named @limit set to 10)

[value] var@: @instance_name []

Routing: Compiler Context.

This method creates an instance value in the instance of the host class.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 var@: @limit`	(Creates an instance variable named @limit initially set to 10)

[] ... ; []

Routing: Compiler Context.

Close off the method definition.

[a_class] .is_class? [true]

Routing: TOS

This method answers true when sent to a class object because classes are classes!

Code	Result
`Object .is_class?`	true

[unspecified a_class] .new [an_instance_of_a_class]

Routing: TOS

Create a new instance of the target class. When the instance of the class is created, the .init method is called on that instance. The unspecified arguments listed above, are the optional arguments to this .init method. The class Object defines a default implementation of the .init method that uses no arguments and takes to no action.

Code	Result
`Object .new`	an_object_instance

[a_class] .parent_class [a_class or nil]

Routing: TOS

Get the parent class of the given class. If there is no parent class (as is the case for the Object class) then return nil.

Code	Result
`Complex .parent_class`	Numeric class
`Object .parent_class`	nil

[a_class] .subclass: subclass_name []

Routing: VM

Create a subclass of the given class. The name of the new class must conform to the follow regex:

```
/^[A-Z][A-Za-z0-9]+$/
```

This means the the name must start with an upper case letter followed by zero or more upper and lower case letters or digits. Note the underscores "_" are not allowed.

This is a helper method of the virtual machine.

Code	Result
`Object .subclass: MyClass`	(Creates the class MyClass, a subclass of Object)
`Object .subclass: wrong`	F10: Invalid class name wrong

[a_class] .to_s [string]

Routing: TOS

Converts the class to a string.

Code	Result
`Object .to_s`	“Object”
`class: MyClass` `MyClass .to_s`	“MyClass”

[] class: class_name []

Routing: VM

This virtual machine helper method is a shortcut for creating new classes. The expression

```
class: MyClass
```

is equivalent to

```
Object .subclass MyClass.
```

The name of the new class must conform to the follow regex:

```
/^[A-Z][A-Za-z0-9]+$/
```

This means the the name must start with an upper case letter followed by one or more upper and lower case letters or digits. Note the underscores “_” are not allowed.

Code	Result
`class: MyClass`	(Creates the class MyClass, a subclass of Object)
`class: wrong`	F10: Invalid class name wrong

Commands

The following are also methods, however, they are designed primarily for interacting directly with the operator in the form of commands. Commands are distinguished by the leading “)” in their name[60].

[a_class])methods []

Routing: TOS

List the active methods defined for this class. Stubs are *not* included in this listing.

```
>Object )methods
Object Shared Methods =
&&              .init           .to_i!          .to_x!          min
)methods        .is_class?      .to_n           <>              nil<>
.               .name           .to_n!          =               nil=
.class          .strlen         .to_r           ^^              not
.clone          .to_f           .to_r!          distinct?       ||
.clone_exclude  .to_f!          .to_s           identical?
.copy           .to_i           .to_x           max
```

[a_class])stubs []

Routing: TOS

List the stub methods defined for this class. Stubs are place holder methods that serve one of two purposes:

1. They are abstract methods in a base class that exist so that a sub-class may replace the stub with an actual method. The stub ensures that the compiler uses the correct routing when using the method.
2. They are sentries for methods in a base class that are not valid to be performed on a particular sub-class. For example, many methods of the Numeric class are not valid in its sub-class Complex.

```
>Object )stubs
Object Shared Stubs =
!       +       0<=     0>      2*      <       >       and     or
)stubs  -       0<=>    0>=     2+      <<      >=      com     xor
*       /       0<>     1+      2-      <=      >>      mod
**      0<      0=      1-      2/      <=>     @       neg
```

60 This convention is a throwback and homage to the command syntax of the APL language on the old PDP-10.

Complex

Inheritance: Complex ← Numeric ← Object

```
Complex Shared Methods =
.cbrt   .e**    .split  .sqrt

Helper Methods =
.to_x   .to_x!  complex

Complex Shared Stubs =
.ceil           .rationalize_to .to_t!          <=>             mod
.emit           .round          <               >
.floor          .to_t           <=              >=
```

A complex number[61] is a number expressed in the form a+bi, where a and b are real numbers and i is the square root of -1. Like other sub-classes of the Numeric class, the Complex class inherits most of its functionality from its parent class. There is one major area where this does not apply. All other Numeric sub-classes are magnitudes. As magnitudes, they may be compared for greater than, less than, etc. While Complex numbers contain magnitudes (accessed via the .magnitude method) they are NOT themselves magnitudes. Thus comparison operations (other than equality or inequality) and many other types of operations are not valid for Complex values.

Complex Literals

Complex literals are supported directly by the compiler. Any number ending in 'i' is considered to be a complex number. The regular expression detecting potential complex numbers is:

```
/\di$/
```

Some example values follow:

Literal[62]	Value[63]
`7i`	0+7i
`-7i`	0-7i
`3.7i`	0+3.7i
`1/2i`	0+1/2i
`-3-7i`	-3-7i

61 See http://en.wikipedia.org/wiki/Complex_number

62 No spaces are permitted within the literal.

63 The real and imaginary parts may be integers, floats or rational numbers. See the respective sections for more details on those types of literals.

`3+7i`	3+7i

Instance Methods

[a_complex] .cbrt [a_complex$^{1/3}$]

Routing: TOS.

Find the cube root of the complex numeric value.

Code	Result
`8+0i .cbrt`	2.0+0.0i

[a_complex] .e** [e^{a_complex}]

Routing: TOS.

Compute the value of e raised to the power of the complex numeric value.

Code	Result
`1+1i .e**`	1.4686939399158851+2.2873552871788423i

[a_complex] .split [real_part imaginary_part]

Routing: TOS.

Split a complex number into its two component parts.

Code	Result
`3+4i .split`	3, 4

[a_complex] .sqrt [a_complex$^{1/2}$]

Routing: TOS.

Find the square root of the complex numeric value.

Code	Result
`2+2i .sqrt`	1.5537739740300374+0.6435942529055827i

[an_object] .to_x [a_complex or nil]

Routing: TOS

This is a helper method of the Object class. Try to convert the object into a Complex. If this is not possible, return nil. Contrast with .to_x!

Code	Result
`5 .to_x`	5+0i
`5.2 .to_x`	5.2+0i
`"5" .to_x`	5+0i
`1+2i .to_x`	1+2i
`"apple" .to_x`	nil

[an_object] .to_x! [a_complex]

Routing: TOS

This is a helper method of the Object class. Try to convert the object into a Complex. If this is not possible, raise an error. Contrast with .to_x

Code	Result
`5 .to_x!`	5+0i
`5.2 .to_x!`	5.2+0i
`"5" .to_x!`	5+0i
`1+2i .to_x!`	1+2i
`"apple" .to_x!`	`Cannot convert a String instance to a Complex instance`

[real_part imaginary_part] complex [a_complex]

Routing: VM

This is a helper method of the Virtual Machine class. Given two numbers, create a complex number. If this cannot be done, an error occurs.

Code	Result
`4 5 complex`	4+5i
`"apple" 5 complex`	F40: Cannot coerce a String instance, Fixnum instance to a Complex

Instance Stubs

A number of methods are stubbed out in the Complex class. All of them are invalid operations because Complex numbers are not magnitudes. These stubbed out methods are:

```
.ceil           .rationalize_to .to_t!          <=>             mod
.emit           .round          <               >
.floor          .to_t           <=              >=
```

Duration

Inheritance: Duration ← Object

```
.intervals .labels

Duration Shared Methods =
*                 .as_years          .to_duration      2+
+                 .days              .to_duration!     2-
-                 .hours             .to_s             2/
.as_days          .largest_interval  .years            f"
.as_hours         .minutes           /                 format
.as_minutes       .months            1+
.as_months        .seconds           1-
.as_seconds       .to_a              2*

Duration Helper Methods =
a_day    a_minute    a_month    a_second    a_year    an_hour

Duration Class Stubs =
.new
```

The Duration class is used to represent a *span* of time. This contrasts with the Time class which represents *points* in time. An analogy can be taken from tennis. The time objects are like the posts on either side of the court, and the duration is like the net spanning the distance between them. Thus when one time value is subtracted from another, the result is the duration, or span of time, between them. This distinction means that Duration objects are different from Time objects in a number of ways:

Firstly: With a time value, the length, in days of a year or month depends on which year or month it is. For example the year 2000 was 366 days long while the year 2001 was 365 days long. With durations this is not the case. Here, years and months cannot be related to any specific year or month, rather they must be tied to a hypothetical *average*[64] year or month. Thus for the Duration class:

1 *year* is 365.2425 days or 365 days, 5 hours, 49 minutes, and 12.0 seconds.

1 *month* is 30.436875 days or 30 days, 10 hours, 29 minutes, and 6.0 seconds.

The definition of days, hours, minutes and seconds exhibit no such complexity[65].

Secondly: When formatting a Time, it is typical to convert months into named months, and days of the week into their name. Since durations are not tied to specific points of time, in a Duration, we only reference the number of months and days, etc. Thus Duration objects have their own formatting specification (see Duration Formatting below).

64 These averages are based on the Gregorian Calendar. See https://en.wikipedia.org/wiki/Gregorian_calendar

65 For now at least, we ignore anomalies like “leap seconds” that are added on various occasions by the Time Lords or some such temporal authorities.

In many ways, Duration objects are also like Numeric data. There will be many cases where we will need to compute durations, count them down etc. Thus the Duration class acts like a Numeric object and can be used in computations. There are however two noteworthy points:

- The Duration class has direct support for only a subset of arithmetic operators: + - * /. For other operations, fear not, the duration is automatically converted into a rational number. If a Duration result is required, the .to_duration method (see below) can be applied to the result.

Code	Result
`a_minute 5 * .class`	Duration
`a_minute .sqr .class`	Rational

- The order of operands matters. Dyadic operators (like +) “bind” to the type of the left most operand. Thus in the example below, the results have different data types. Again, the .to_duration method (see below) can be applied to the result.

Code	Result
`a_minute 12 + .class`	Duration
`12 a_minute + .class`	Fixnum[66]

Note: If in doubt, the .to_duration method (see below) can be safely applied to a duration value with no ill effects and very little time or effort.

Creating Duration Values

Duration objects are not created in the typical fashion. In fact the standard object creating method, .new is stubbed out and not available. None-the-less there are several ways to create Duration objects:

- The difference of two time objects is a Duration equal to the span of time between them.
- The predefined methods a_day, a_minute, etc create Duration objects with the appropriate span value. See special duration values below for more details.
- The .to_duration method allows a simple number to be converted into a Duration with a span set to the number of seconds of the number.
- The .to_duration method also works with an array of numbers. These values are interpreted as [years months days hours minutes seconds]. If fewer than six data are present, leading values are assumed to be zero. So the following are equivalent:

```
[ 1 ] .to_duration
[ 0 0 0 0 0 1 ] .to_duration.
```

66 Conversion to another type can have other side-effects. For example, conversion to Fixnum could result in the loss of the fractions of seconds of the original duration. Similarly, conversion to a float can also result in loss of accuracy.

Special Duration Values

[] a_day [a_duration]

Routing: VM

Push a duration with a span of a day onto the stack.

Code	Result
`a_day`	Duration instance <86400.0 seconds>

[] a_minute [a_duration]

Routing: VM

Push a duration with a span of a minute onto the stack.

Code	Result
`a_minute`	Duration instance <60.0 seconds>

[] a_month [a_duration]

Routing: VM

Push a duration with the average span of a month onto the stack.

Code	Result
`a_month`	Duration instance <2629746.0 seconds>

[] a_second [a_duration]

Routing: VM

Push a duration with a span of a second onto the stack.

Code	Result
`a_second`	Duration instance <1.0 seconds>

[] a_year [a_duration]

Routing: VM

Push a duration with the average span of a year onto the stack.

Code	Result
`a_year`	Duration instance <31556952.0 seconds>

[] an_hour [a_duration]

Routing: VM

Push a duration with a span of an hour onto the stack.

Code	Result
`an_hour`	Duration instance <3600.0 seconds>

Duration Formatting

The formatting facility for Duration objects is derived from the facilities of the Ruby format_engine gem[67]. A format string is a string with optional text and zero or more format sequences. The structure of a format sequence is shown below with elements in brackets representing optional components.

```
%[flags][sign][width][.precision]type
```

The type parameter is a single character that describes the data within the duration being formatted. The following are supported:

Component Format Types

Type	Format Description
d	Whole days in the month.
D	Total (with fractional) days.
h	Whole hours in the day.
H	Total (with fractional) hours.
m	Whole minutes in the hour.
M	Total (with fractional) minutes.
o	Whole months in the year.
O	Total (with fractional) months.

67 For more details please see: https://rubygems.org/gems/format_engine

Type	Format Description
s	Total (with fractional) seconds in the minute.
S	Total (with fractional) seconds.
y	Whole years.
Y	Total (with fractional) years.

Note: Formats with a “whole” attribute do not support the precision option. Formats with a “fractional” attribute, default to six digits of precision.

Other Format Types

Type	Format Description
B	Brief summary format. The total (with fractional) of the largest, non-zero time unit.
f	Raw float format. Total seconds in floating point format.
r	Raw rational format. Total seconds in rational format.

Note: Currently f and r formats do not have label support (See the $ option below)

Format Sign

Type	Format Description
+	(The default) The output is right justified within the format width.
-	The output is left justified within the format width.

Format Flags

Type	Format Description
?	Suppress output if the value, or the value for this label (see $ below) is zero.
$	Output the text label appropriate for the value. For example whereas %y outputs the year, %$y outputs the text “years” (or “year” if there is exactly one year in the duration object being formatted).

Note: If both the ? and $ options are used, the ? flag must come first in the format string.

Examples

The examples that follow all use the f"format string" method instead of the "format string" format method. The first form is shorter, and often clearer. The second form must be used when it is desired to compute or lookup the format string.

This batch of formats are demonstrated with the shared input of “123456 .to_duration”. This common input is not shown for brevity.

Code	Result
`f"About %4.1B%$B"`	`"About 1.4 days"`
`f"%d %h %1.0s"`	`"1 10 36"`
`f"%d%$d %h%$h %s%$s"`	`"1 day 10 hours 36.000000 seconds"`
`f"%d%$d %h%$h %1.0s%$s"`	`"1 day 10 hours 36 seconds"`
`f"%d%$d %h%$h %3.1s%$s"`	`"1 day 10 hours 36.0 seconds"`
`f"%3.1D%$D"`	`"1.4 days"`
`f"%3.1H%$H"`	`"34.3 hours"`
`f"%3.1M%$M"`	`"2057.6 minutes"`
`f"%3.1S%$S"`	`"123456.0 seconds"`
`f"%r"`	`"123456/1"`
`.to_s`	`"Duration instance <123456.0 seconds>"`

Class Methods

[Duration] .intervals [an_array]

Routing: TOS

Push an array of durations onto the stack. This array contains durations corresponding to a year, a month, a day, an hour, a minute, and a second.

Code	Result
`Duration .intervals`	[Duration instance <31556952.0 seconds> Duration instance <2629746.0 seconds> Duration instance <86400.0 seconds> Duration instance <3600.0 seconds> Duration instance <60.0 seconds> Duration instance <1.0 seconds>]

[Duration] .labels [an_array]

Routing: TOS

Push an array of interval labels onto the stack. These correspond to a year, a month, a day, an hour, a minute, and a second.

Code	Result
`Duration .labels`	[“years” “months” “days” “hours” “minutes” “seconds”]

Instance Methods

[a_duration a_duration or a_number] * [a_duration]

Routing: NOS

This is the multiplication operator for the Duration class.

Code	Result
`a_minute 5 *`	Duration instance <300.0 seconds>

[a_duration a_duration or a_number] + [a_duration]

Routing: NOS

This is the addition operator for the Duration class.

Code	Result
`a_minute 5 +`	Duration instance <65.0 seconds>

[a_duration a_duration or a_number] - [a_duration]

Routing: NOS

This is the subtraction operator for the Duration class.

Code	Result
`a_minute 5 -`	Duration instance <55.0 seconds>

[a_duration] .as_days [a_float]

Routing: TOS

Convert a duration to a float representing the number of days (including fractions) in the span.

Code	Result
`a_month .as_days f"%4.2f"`	“30.44”
`500000 .to_duration .as_days f"%4.2f"`	“5.79”

[a_duration] .as_hours [a_float]

Routing: TOS

Convert a duration to a float representing the number of hours (including fractions) in the span.

Code	Result
`a_day .as_hours f"%4.2f"`	“24.00”
`500000 .to_duration .as_hours f"%4.2f"`	“138.89”

[a_duration] .as_minutes [a_float]

Routing: TOS

Convert a duration to a float representing the number of minutes (including fractions) in the span.

Code	Result
`an_hour .as_minutes f"%4.2f"`	“60.00”
`50000 .to_duration .as_minutes f"%4.2f"`	“833.33”

[a_duration] .as_months [a_float]

Routing: TOS

Convert a duration to a float representing the number of months (including fractions) in the span.

Code	Result
`a_year 3 * .as_months f"%4.2f"`	“36.00”
`50000 .to_duration .as_months f"%4.2f"`	“0.0190”

[a_duration] .as_seconds [a_float]

Routing: TOS

Convert a duration to a float representing the number of seconds (including fractions) in the span.

Code	Result
`a_day 11.0 / .as_seconds f"%4.2f"`	“7854.55”
`50000 .to_duration .as_seconds f"%4.2f"`	“50000.00”

[a_duration] .as_years [a_float]

Routing: TOS

Convert a duration to a float representing the number of years (including fractions) in the span.

Code	Result
`a_year pi * .as_years f"%6.4"`	“3.1416”
`3.0E9 .to_duration .as_years f"%4.2f"`	“95.07”

[a_duration] .days [an_integer]

Routing: TOS

Extract the number of whole days within the month of the duration's span.

Code	Result
`a_day 40 * .days`	9
`50000 .to_duration .days`	0

[a_duration] .hours [an_integer]

Routing: TOS

Extract the number of whole hours within the day of the duration's span.

Code	Result
`an_hour 40 * .hours`	16
`50000 .to_duration .hours`	13

[a_duration] .largest_interval [0..5]

Routing: TOS

Return the index of the largest non-zero interval unit within the span of the duration. This is zero for years, one for months, two for days, three for hours, four for minutes, and five for seconds. These index values correspond to the indexes for these intervals in the Duration class methods ".intervals" and ".labels" (see above).

Note: if the span is less than one second, an index of five is returned for the fractions of seconds in the span.

Code	Result
`a_year .largest_interval`	0
`a_month .largest_interval`	1
`a_day .largest_interval`	2
`an_hour .largest_interval`	3
`a_minute .largest_interval`	4
`a_second .largest_interval`	5
`0 .to_duration .largest_interval`	5

[a_duration] .minutes [an_integer]

Routing: TOS

Extract the number of whole minutes within the hour of the duration's span.

Code	Result
`an_hour 1- .minutes`	59
`50000 .to_duration .minutes`	53

[a_duration] .months [an_integer]

Routing: TOS

Extract the number of whole months within the year of the duration's span.

Code	Result
`a_month 16 * .months`	4
`1.0E7 .to_duration .months`	3

[a_duration] .seconds [a_float]

Routing: TOS

Extract the number of seconds (with fractions) within the minute of the duration's span.

Code	Result
`pi 20 * .to_duration .seconds`	2.8318530717958623
`50000 .to_duration .seconds`	20.0

[a_duration] .to_a [an_array]

Routing: TOS

Convert the duration into an array where the elements represent the years, months, days, hours, minutes, and seconds of the duration's span. These are all integers except for the seconds which is a float.

Code	Result
`a_year 2* 1- .to_a`	[1 11 30 10 29 5.0]
`50000 .to_duration .to_a`	[0 0 0 13 53 20.0]
`pi .to_duration .to_a`	[0 0 0 0 0 3.141592653589793]

[a_duration or a_number or an_array] .to_duration [a_duration or nil]

Routing: TOS

This method is a composite of a method and some helpers. Together they implement a protocol for converting data into Duration instances. If the conversion is unable to proceed, the value nil is returned instead of a duration.

Code	Result
`50000 .to_duration`	Duration instance <50000.0 seconds>
`an_hour .to_duration`	Duration instance <3600.0 seconds>
`[ 50000 ] .to_duration`	Duration instance <50000.0 seconds>
`[ 1 2 3 4 5 6 ] .to_duration`	Duration instance <37090350.0 seconds>
`[ 1 2 3 4 5 6 7 ] .to_duration`	nil
`"apple" .to_duration`	nil
`3+4i .to_duration`	nil

[a_duration or a_number or an_array] .to_duration! [a_duration]

Routing: TOS

This method is a composite of a method and some helpers. Together they implement a protocol for converting data into Duration instances. If the conversion is unable to proceed, an error is raised.

Code	Result
`50000 .to_duration!`	Duration instance <50000.0 seconds>
`an_hour .to_duration!`	Duration instance <3600.0 seconds>
`[ 50000 ] .to_duration!`	Duration instance <50000.0 seconds>
`[ 1 2 3 4 5 6 ] .to_duration!`	Duration instance <37090350.0 seconds>
`[ 1 2 3 4 5 6 7 ] .to_duration!`	F40: Cannot convert Array instance to a Duration instance
`"apple" .to_duration!`	F40: Cannot convert String instance to a Duration instance
`3+4i .to_duration!`	F40: Cannot convert Complex instance to a Duration instance

[a_duration] .to_s [a_string]

Routing: TOS

The default conversion to string. Mostly for debugging etc.

Code	Result
`an_hour .to_s`	“Duration instance <3600.0 seconds>”

[a_duration] .years [an_integer]

Routing: TOS

Extract the number of whole years within the year of the duration's span.

Code	Result
`a_year 3/2 * .years`	1
`50000 .to_duration .years`	0

[a_duration a_duration or a_number] / [a_duration]

Routing: NOS

This is the division operator for the Duration class.

Code	Result
`a_year 2 / .to_a`	[0 6 0 0 0 0.0]

[a_duration] 1+ [a_duration]

Routing: TOS

Add one second to the duration. Note: This method does *not* mutate the original duration.

Code	Result
`a_minute 1+`	Duration instance <61.0 seconds>

[a_duration] 1- [a_duration]

Routing: TOS

Subtract one second from the duration. Note: This method does *not* mutate the original duration.

Code	Result
`a_minute 1-`	Duration instance <59.0 seconds>

[a_duration] 2* [a_duration]

Routing: TOS

Double the span of the duration. Note: This method does *not* mutate the original duration.

Code	Result
`a_minute 2*`	Duration instance <120.0 seconds>

[a_duration] 2+ [a_duration]

Routing: TOS

Add two seconds to the duration. Note: This method does *not* mutate the original duration.

Code	Result
`a_minute 2+`	Duration instance <62.0 seconds>

[a_duration] 2- [a_duration]

Routing: TOS

Subtract two seconds from the duration. Note: This method does *not* mutate the original duration.

Code	Result
`a_minute 2-`	Duration instance <58.0 seconds>

[a_duration] 2/ [a_duration]

Routing: TOS

Halve the span of the duration. Note: This method does *not* mutate the original duration.

Code	Result
`a_minute 2/`	Duration instance <30.0 seconds>

[a_duration] f"a format string" [a_string]

Routing: NOS

The duration short form formatted conversion to a string. See Duration Formatting above for more details.

[a_duration a_string] format [a_string]

Routing: NOS

The duration long form formatted conversion to a string. See Duration Formatting above for more details.

FalseClass

Inheritance: FalseClass ← Object

```
FalseClass Shared Methods =
&&      ^^      not     ||

Helper Methods =
false
```

The FalseClass is the class behind the value false. The value false is used to process a number of Boolean oriented operations. The NilClass serves as a surrogate false value and duplicates the functionality of false.

FalseClass Literals

Instances of the class FalseClass are available though the Virtual Machine helper method “false”.

Instance Methods

[false object] && [false]

Routing: NOS.

Logical AND for the case where the first operand is false. Always false.

Note: Other parts of this method are implemented in the Object and NilClass classes.

Code	Result
`false false &&`	false
`false true &&`	false
`true false &&`	false
`true true &&`	true

[false object] ^^ [true or false]

Routing: NOS.

Logical, exclusive OR for the case where the first operand is false. This takes on the value of the second operand converted to a Boolean value.

Note: Other parts of this method are implemented in the Object and NilClass classes.

Code	Result
`false false ^^`	false
`false true ^^`	true
`true false ^^`	true
`true true ^^`	false

[false object] || [true or false]

Routing: NOS.

Logical, inclusive OR for the case where the first operand is false. This takes on the value of the second operand converted to a Boolean value.

Note: Other parts of this method are implemented in the Object and NilClass classes.

Code	Result
`false false ||`	false
`false true ||`	true
`true false ||`	true
`true true ||`	true

Float

Inheritance: Float ← Numeric ← Object

```
Float Shared Methods =
.to_r  .to_r!

Helper Methods =
.to_f   .to_f!
```

Float[68] or floating point data are an approximation of the mathematical set of Real numbers. In fOOrth, this approximation is based on the IEEE-754[69] Double Precision data type. The Float class inherits its functionality from the Numeric class.

Float Literals

Like other numeric literals, float literals are implemented directly by the parser. Any number with an embedded '.' is considered to be a float. The regular expression that detects potential float point numbers is:

```
/\d\.\d/
```

Some example values follow:

Literal[70]	Value
`7.0`	7.0
`7.0E3`	7000.0
`7.0E-3`	0.007
`-7.0`	-7.0
`-7.0E3`	-7000.0
`-7.0E-3`	-0.007

68 See http://en.wikipedia.org/wiki/Floating_point
69 See http://en.wikipedia.org/wiki/IEEE_floating_point
70 No spaces are permitted within the literal.

Instance Methods

[an_object] .to_f [a_float or nil]

Routing: TOS

Try to convert the object to a float. If this is not possible, return nil. Contrast with .to_f! This is a helper method of the Object class.

Code	Result
`"43.1" .to_f`	43.1
`99 .to_f`	99.0
`"apple" .to_f`	nil

[an_object] .to_f! [a_float]

Routing: TOS

Try to convert the object to a float. If this is not possible raise an error. Contrast with .to_f This is a helper method of the Object class.

Code	Result
`"43.1" .to_f!`	43.1
`99 .to_f!`	99.0
`"apple" .to_f!`	`Cannot coerce a String instance to a Float instance`

[a_float] .to_r [a_rational or nil]

Routing: TOS

Try to convert the float into a Rational. If this is not possible, return nil. Contrast with .to_r! This method replaces the default implementation in the Object class to produce better results for floating point data.

Code	Result
`2.5 .to_r`	5/2
`1.3 .to_r`	13/10
`infinity .to_r`	nil

[a_float] .to_r! [a_rational]

Routing: TOS

Try to convert the float into a Rational. If this is not possible, raise an error. Contrast with .to_r! This method replaces the default implementation in the Object class to produce better results for floating point data.

Code	Result
`2.5 .to_r!`	5/2
`1.3 .to_r!`	13/10
`infinity .to_r!`	`Cannot convert a Float instance to a Rational instance`

Hash

Inheritance: Hash ← Object

```
>Hash )methods
Hash Shared Methods =
.[]!      .each{{   .keys     .pp       .to_s
.[]@      .empty?   .length   .strmax2  .values

Helper Methods =
{
```

Hash[71] objects are collections of data indexed by a value. This can be a number, a string or any other sort of value. The fOOrth language system supports the creation of hash literals and has several methods for putting data into and pulling data out of hashes.

Hash Literals

Hash literals are supported by the virtual machine method "{" and the locally defined methods "->" and "}". The general usage is:

```
{ (key/value generating code goes here) }
```

Where the data generation code is code that deposits zero or more key value pairs onto the stack. The opening "{" creates an empty hash. The "->" method takes a key and a value from the stack and adds this key/value pair to the hash. When the closing "}" is encountered, the operation is wrapped up. Here are some illustrations of hash literals in action:

```
>{ } .
{ }
>{ 1 2 ->  2 3 ->  3 5 ->  4 7 ->  5 11 -> } .
{ 1 2 -> 2 3 -> 3 5 -> 4 7 -> 5 11 -> }
```

Some points of interest:

- The first example creates an "empty" hash with zero data elements.
- Hash key and data do NOT need to be the same "type" of data. Mixing is allowed.
- Any statements that generate key/data pairs are permitted. Consider:

```
>{ 0 10 do i i -> loop } .
{ 0 0 -> 1 1 -> 2 2 -> 3 3 -> 4 4 -> 5 5 -> 6 6 -> 7 7 -> 8 8 -> 9 9 -> }
>{ 0 10 do i i dup * -> loop } .
{ 0 0 -> 1 1 -> 2 4 -> 3 9 -> 4 16 -> 5 25 -> 6 36 -> 7 49 -> 8 64 -> 9 81 -> }
```

71 Please see http://en.wikipedia.org/wiki/Hash_table for more information.

Hash Literal Methods

[] { [a_hash]

Routing: VM

This method begins the creation of a hash literal value.

Code	Result
`{`	{}

Local Methods:

[a_hash a_key a_value] -> [a_hash]

Routing: Compiler Context.

This method takes a key and a value and adds it to the hash.

Code	Result
`{ "a" 1 ->`	{“a”=>1}

[a_hash] … } [a_hash]

Routing: Compiler Context.

This method closes off the context of the hash literal creation.

Code	Result
`{ "a" 1 -> }`	{“a”=>1}

Hash Literals in Action

The following shows the action of the code 1 2 { 3 4 -> 5 6 -> } with)show and)debug active:

```
>1 2
Tags=[:numeric] Code="vm.push(1); "
Tags=[:numeric] Code="vm.push(2); "

[ 1 2 ]
>{
Tags=[:immediate] Code="vm._224(vm); "
  nest_context
  Code="vm.push(Hash.new); "

[ 1 2 { } ]
>>3 4 ->
Tags=[:numeric] Code="vm.push(3); "
Tags=[:numeric] Code="vm.push(4); "
```

```
Tags=[:immediate] Code="vm.context[:_315].does.call(vm); "
  Code="vm.add_to_hash; "

[ 1 2 { 3 4 -> } ]
>>5 6 ->
Tags=[:numeric] Code="vm.push(5); "
Tags=[:numeric] Code="vm.push(6); "
Tags=[:immediate] Code="vm.context[:_315].does.call(vm); "
  Code="vm.add_to_hash; "

[ 1 2 { 3 4 -> 5 6 -> } ]
>>}
Tags=[:immediate] Code="vm.context[:_316].does.call(vm); "
  unnest_context
  Code=""

[ 1 2 { 3 4 -> 5 6 -> } ]
```

Instance Methods

[value index hash] .[]! []

Routing: TOS

Store the specified value at the index of the hash.

Note: This method *does* mutate the hash.

Code	Result
`"Hello" 5 $myhash .[]!`	(“Hello” is stored with key 5 in myhash.)

[index hash] .[]@ [value]

Routing: TOS

Retrieve the value stored in the hash at the specified index. If the index does not correspond to a location within the hash, the value nil is returned instead.

Code	Result
`1  { 1 2 -> 3 4 -> } .[]@`	2
`11 { 1 2 -> 3 4 -> } .[]@`	nil

[a_hash] .each{{ … }} [unspecified]

Routing: NOS (since the Procedure Literal is TOS)

This method is the hash item iterator. It processes each element of the hash in turn, calling the embedded procedure literal block with the value (v) and index (x) of the current array item.

For more information on the methods local to the embedded procedure, see the Procedure class.

Code	Result
`{ 0 "1" -> 1 "2" -> 2 "3" -> 3 "4" -> } .each{ v x 1+ * . space }`	(Prints out:) 1 22 333 4444
`{ 0 "1" -> 1 "2" -> 2 "3" -> 3 "4" -> } .each{ v . space }`	(Prints out:) 1 2 3 4
`{ 0 "1" -> 1 "2" -> 2 "3" -> 3 "4" -> } .each{ x . space }`	(Prints out:) 0 1 2 3

[a_hash] .empty? [a_boolean]

Routing: TOS

Is this hash devoid of key/value pairs?

Code	Result
`{ } .empty?`	true
`{ "a" 1 -> } .empty?`	false

[a_hash] .keys [an_array]

Routing: TOS

This method gathers up the keys in a hash and places them in an array.

Note: This method does *not* mutate the hash.

Code	Result
`{ 1 2 -> 3 4 -> } .keys`	[1 3]

[a_hash] .length [a_count]

Routing: TOS

How many key/value pairs are in this hash?

Code	Result
`{ } .length`	0
`{ "a" 1 -> } .length`	1

[hash] .pp []

Routing: TOS

This method is a pretty printer for hashes. The data in the hash is displayed with in columns for an 80 character wide display and a blank line every 50 lines. The primary use of this method was in preparing the lists of method names used in this guide. No value is returned.

Code	Result
`{ 1 2 -> 4 555 -> } .pp`	Displays “1=>2 4=>555”

[a_hash] .to_s [a_string]

Routing: TOS

Convert the hash to a string representation of that hash.

Code	Result
`{ 1 2 -> 3 4 -> } .to_s`	“{ 1 2 -> 3 4 -> }”

[hash] .strmax2 [width]

Routing: TOS

Given an hash, this method determines the width of the largest string representation of the keys and of the values. This method is a helper method for the .pp pretty print method.

Note: This method does *not* mutate the original hash.

Code	Result
`{ 1 2 -> 4 555 -> } .strmax2`	1 3

[a_hash] .values [an_array]

Routing: TOS

This method gathers up the values in a hash and places them in an array.

Note: This method does *not* mutate the hash.

Code	Result
`{ 1 2 -> 3 4 -> } .values`	[2 4]

InStream

Inheritance: InStream ← Object

```
InStream Class Methods =
.get_all .open    .open{{

InStream Shared Methods =
.close .getc  .gets  ~getc  ~gets

InStream Class Stubs =
.new
```

The InStream class is used to support the reading of information from text files in an accessible file system.

Class Methods

[file_name InStream] .get_all [[“line 1”, … “line N”]]

Routing: TOS

This method opens the file of the given name for reading, reads the entire file into an array of lines of text in that array, then closes the file.

Note: If the file being read is large, a large amount of data will be read.

Code	Result
`"test.txt" InStream .get_all`	[“Line 1”, “Line 2”, “Line 3”]
`"bad.txt" InStream .get_all`	F50: Unable to open the file bad.txt for reading all.

[file_name InStream] .open [an_instream]

Routing: TOS

This method opens the file of the given name for reading. It returns an instance of a InStream object that may be used for reading data from that file. The programmer is responsible for managing that instance at all times. A typical strategy is to use a local value to hold this instance.

Note: The programmer is responsible for ensuring that the file object is finally closed.

Code	Result
`"test.txt" InStream .open val: rf`	(Creates a local value rf with an InStream instance.)
`"bad.txt" InStream .open`	F50: Unable to open the file bad.txt for reading.

[file_name InStream] .open{{ … }} [unspecified]

Routing: NOS (since the Procedure Literal is TOS)

This is actually a Virtual Machine method proxy for InStream. This method opens the file of the given name for reading, it then executes the embedded procedure literal block (between the {{ and }}) with self set to the opened file for the duration of the block. Finally, it closes the file.

For more information on the methods local to the embedded procedure, see the Procedure class.

Notes: The InStream instance is automatically (and always) closed at the end of the code block.

Code	Result
`"test.txt" InStream .open{ ~gets }`	“Line 1”

Instance Methods

[an_instream] .close []

Routing: TOS

This method closes of the file associated with the InStream instance.

Code	Result
`fv .close`	(Closes the file stored in the value fv. See .open above)

[an_instream] .getc [a_character]

Routing: TOS

This method reads a single character from the file connected to an instance of an InStream.

Code	Result
`fv .getc`	“L”

[an_instream] .gets [a_string]

Routing: TOS

This method reads a line of text from the file connected to an instance of an InStream.

Code	Result
`fv .getc`	“Line 1”

[] ~getc [a_character]

Routing: Self

This method reads a character of text from the file connected to an instance of an InStream that is the current “self” value. This most typically happens in the code block of an .open{ … } method or an exclusive method added to an InStream instance.

Code	Result
`"test.txt" InStream .open{ ~getc }`	“L”

[] ~gets [a_string]

Routing: Self

This method reads a line of text from the file connected to an instance of an InStream that is the current “self” value. This most typically happens in the code block of an .open{ … } method or an exclusive method added to an InStream instance.

Code	Result
`"test.txt" InStream .open{ ~gets }`	“Line 1”

Class Stubs

The following method is stubbed out in the InStream class and not available: .new

Integer

Inheritance: Integer ← Numeric ← Object

```
Integer Shared Methods =
.even? .join  .odd?  2/     >>     com    xor
.gcd   .lcm   2*     <<     and    or

Helper Methods
.to_i   .to_i!
```

Integers[72] are numbers that may be represented without any division or fractional components. In fOOrth, integers behave much more like the abstract integers of mathematics than those traditionally associated with computers. Unlike common computer integers[73], fOOrth integers have no preset capacity or limit. They are able to expand to accommodate data as needed without concern for issues such as overflow. It is however true, that given a finite computer memory sub-system, such expansion cannot go on without limit, still the limit is a rather ginormous.

Integer Literals

Like other numeric literals, integer literals are implemented directly by the parser. The parser does not have a specific rule for parsing integer literals. The parser will attempt to parse a language token as an integer when all other avenues of parsing have failed. Thus, like FORTH, in fOOrth integer literals are the parse of last resort.

Some examples follow:

Literal[74]	Value
`7`	7
`-7`	-7
`445556678789933`	445556678789933
`0xff`	255
`0xfffffffff`	1099511627775
`0xDeadBeef`	3735928559

72 See http://en.wikipedia.org/wiki/Integer
73 See http://en.wikipedia.org/wiki/Integer_(computer_science)
74 No spaces are permitted within the literal.

Instance Methods

[an_integer] .even? [a_boolean]

Routing: TOS

This method returns true if the integer is even and false if it is odd.

Code	Result
`2 .even?`	true
`3 .even?`	false

[an_integer an_integer] .gcd [an_integer]

Routing: TOS

This method computes the greatest common divisor of the two integers.

Code	Result
`50 6 .gcd`	2
`2345 2890 .gcd`	5

[a_1 a2 … a_N N] .join [after]

Routing: TOS

Join the top N stack elements into an array. See the Array class for more details.

[an_integer an_integer] .lcm [an_integer]

Routing: TOS

This method computes the lowest common denominator of the two integers.

Code	Result
`50 6 .lcm`	150
`9 15 .lcm`	45
`-5 12 .lcm`	60

[an_integer] .odd? [a_boolean]

Routing: TOS

This method returns true if the integer is odd and false if it is even.

Code	Result
`2 .odd?`	false
`3 .odd?`	true

[an_object] .to_i [an_integer or nil]

Routing: TOS

Try to convert the object to an integer. If this is not possible, return nil. Contrast with .to_i! This is a helper method of the Object class.

Code	Result
`"43.1" .to_i`	43
`99 .to_i`	99
`"apple" .to_i`	nil

[an_object] .to_i! [an_integer]

Routing: TOS

Try to convert the object to an integer. If this is not possible raise an error. Contrast with .to_i! This is a helper method of the Object class.

Code	Result
`"43.1" .to_i!`	43
`99 .to_i!`	99
`"apple" .to_i!`	`Cannot coerce a String instance to an Integer instance`

[an_integer] 2* [an_integer]

Routing: TOS

Double the value of the integer.

Code	Result
`13 2*`	26

[an_integer] 2/ [an_integer]

Routing: TOS

Halve the value of the integer. Note that integer division is employed with rounding down towards nagative infinity.

Code	Result
`7 2/`	3
`-7 2/`	-4

[an_integer an_integer] << [an_integer]

Routing: NOS

Shift the integer value left by the number of positions indicated. A shift count of zero takes no action. A negative shift count shifts right by the number of positions indicated.

Note: When shifting right, rounding toward negative infinity is used.

Code	Result
`4 14 <<`	65536
`4 0 <<`	4
`4 -1 <<`	2
`-10 3 <<`	-80
`-10 5 <<`	-320
`-10 -3 <<`	-2
`-10 -5 <<`	-1

[an_integer an_integer] >> [an_integer]

Routing: NOS

Shift the integer value right by the number of positions indicated. A shift count of zero takes no action. A negative shift count shifts left by the number of positions indicated.

Note: When shifting right, rounding toward negative infinity is used.

Code	Result
`4 1 >>`	2
`4 0 >>`	4
`4 -1 >>`	8
`-10 3 >>`	-2
`-10 5 >>`	-1
`-10 -3 >>`	-80
`-10 -5 >>`	-320

[an_integer an_integer] and [an_integer]

Routing: NOS

Compute the bit-wise and function of the two integer values.

Code	Result
`15 40 and`	8

[an_integer] com [an_integer]

Routing: TOS

This method computes the bit-wise complement of the integer value.

Code	Result
`34 com`	-35
`0 com`	-1

[an_integer an_integer] or [an_integer]

Routing: NOS

Compute the bit-wise inclusive or function of the two integer values.

Code	Result
`15 40 or`	47

[an_integer an_integer] xor [an_integer]

Routing: NOS

Compute the bit-wise exclusive or function of the two integer values.

Code	Result
`15 40 xor`	39

Mutex

Inheritance: Mutex ← Object

```
Mutex Class Methods =
.do{{

Mutex Shared Methods =
.do{{   .lock   .unlock
```

The Mutex class is used to support the concept of mutual exclusion[75]. Mutual exclusion refers to the requirement of ensuring that no two concurrent processes are in their critical section at the same time; it is a basic requirement in concurrency control, to prevent race conditions.

The default implementation of .new is used to create new instances of Mutex as follows:

```
Mutex .new
```

An example use of the Mutex is coordinating access to a counter shared by many threads[76]:

```
( a_count test_mutex 100*a_count )
: test_mutex
  0 var: ctr
  Array .new{{
    x .to_s Thread .new{{
      0 100 do
        Mutex .do{{ ctr @ 1+ ctr ! }}
        0.001 .sleep
      loop
    }}
  }} .each{{ v .join }}
  ctr @ ;
```

Class Methods

[Mutex] .do{{ … }} []

Routing: NOS (since the Procedure Literal is TOS)

This method executes the embedded procedure literal block with an exclusion for all other code blocks guarded by a shared, system wide mutex instance. See example above.

For more information on the methods local to the embedded procedure, see the Procedure class.

75 See https://en.wikipedia.org/wiki/Mutual_exclusion

76 This example may be found in mutex.foorth which may be found in the doc/snippets folder.

Instance Methods

[a_mutex] .do{{ … }} []

Routing: NOS (since the Procedure Literal is TOS)

This method executes the embedded procedure literal block with an exclusion for all other code blocks guarded by the same mutex instance. The above example, rewritten to use a local mutex instance is shown below:

```
( a_count test_mutex 100*a_count )
: test_mutex
  0 var: ctr  Mutex .new val: mex
  Array .new{{
    x .to_s Thread .new{{
      0 100 do
        mex .do{{ ctr @ 1+ ctr ! }}
        0.001 .sleep
      loop
    }}
  }} .each{{ v .join }}
  ctr @ ;
```

For more information on the methods local to the embedded procedure, see the Procedure class.

[a_mutex] .lock []

Routing: TOS

Gain a lock on a resource, waiting until the mutex is unlocked before proceeding.

Code	Result
`mex .lock`	

[a_mutex] .unlock []

Routing: TOS

Unlock the mutex, giving access to the resource to other threads.

Code	Result
`mex .unlock`	

NilClass

Inheritance: NilClass ← Object

```
NilClass Shared Methods =
&&    ^^    nil<>   nil=  not   ||
```

The value nil of the class NilClass is a placeholder for nothing, that is the absence of all data. In other languages like “C” NULL is a special pointer value with restrictions on its use. In fOOrth, nil is just another object with a very bad reputation!

Note that in Boolean expressions, nil is treated as an alias for false.

NilClass Literals

Instances of the NilClass are available through the Virtual Machine helper method “nil”.

Instance Methods

[nil object] && [false]

Routing: NOS.

Logical AND for the case where the first operand is nil. Always false.

Note: Other parts of this method are implemented in the Object and FalseClass classes.

Code	Result
`nil true &&`	false

[nil object] ^^ [true or false]

Routing: NOS.

Logical, exclusive OR for the case where the first operand is nil. This takes on the value of the second operand converted to a Boolean value.

Note: Other parts of this method are implemented in the Object and FalseClass classes.

Code	Result
`nil false ^^`	false
`nil true ^^`	true

[nil object] || [true or false]

Routing: NOS.

Logical, inclusive OR for the case where the first operand is nil. This takes on the value of the second operand converted to a Boolean value.

Note: Other parts of this method are implemented in the Object and FalseClass classes.

Code	Result
`nil false ||`	false
`nil true ||`	true

Numeric

Inheritance: Numeric ← Object

```
Numeric Shared Methods =
*               .atanh          .ln             .sqr            1-
**              .c2p            .log10          .sqrt           2*
+               .cbrt           .log2           .tan            2+
-               .ceil           .magnitude      .tanh           2-
.1/x            .conjugate      .numerator      .to_t           2/
.10**           .cos            .p2c            .to_t!          <
.2**            .cosh           .polar          /               <=
.abs            .cube           .r2d            0<              <=>
.acos           .d2r            .rationalize_to 0<=             >
.acosh          .denominator    .real           0<=>            >=
.angle          .e**            .round          0<>             mod
.asin           .emit           .round_to       0=              neg
.asinh          .floor          .sin            0>
.atan           .hypot          .sinh           0>=
.atan2          .imaginary      .sleep          1+

Helper Methods =
.to_n        .to_n!
```

The Numeric class is the abstract base class for numeric data. It is the location for the vast majority of methods that act on such data. In use, data will be instances of Complex, Float, Integer (via Bignum and Fixnum), and Rational data.

Special Numeric Values

[] -infinity [a_float]

Routing: VM

This method pushes the special floating point value -Infinity.

Code	Result
`-infinity`	-Infinity

[] dpr [a_float]

Routing: VM

This method pushes the degrees per radians constant onto the stack. This has a value of 180/pi.

Code	Result
`dpr`	57.29577951308232

[] e [a_float]

Routing: VM

This method pushes the value e, the base of the natural logarithms, onto the stack.

Code	Result
`e`	2.718281828459045

[] epsilon [a_float]

Routing: VM

This smallest float such that 1.0 + epsilon <> 1.0. The more generalized case is

```
n + (n * epsilon) ≠ n
```

Code	Result
`epsilon`	2.220446049250313e-16
`1.0 epsilon +`	1.0000000000000002
`1.0 epsilon 2/ +`	1.0
`100.0 dup epsilon * +`	100.00000000000003
`0.01 dup epsilon * +`	0.010000000000000002

[] infinity [a_float]

Routing: VM

This method pushes the special floating point value Infinity.

Code	Result
`-infinity`	Infinity

[] max_float [a_float]

Routing: VM

This method pushes the value of the largest allowed floating point number, currently this is 1.7976931348623157e+308 on the current test environment.

Code	Result
`max_float`	1.7976931348623157e+308

[] min_float [a_float]

Routing: VM

This method pushes the value of the smallest, non-zero, floating point number, currently this is 2.2250738585072014e-308 on the current test environment.

Code	Result
`min_float`	2.2250738585072014e-308

[] nan [a_float]

Routing: VM

This method pushes the special floating point value Not-A-Number (NaN).

Code	Result
`nan`	NaN

[] pi [a_float]

Routing: VM

This method pushes the famous value pi, the ratio of a circle's circumference to its diameter, onto the stack.

Code	Result
`pi`	3.1415926535897937[7]

77 In 1897, several attempts were made to legislate the value of pi to a number of incorrectly computed values. Fortunately, these efforts were unsuccessful.

Instance Methods

[a_numeric a_numeric] * [a_numeric]

Routing: NOS

The multiplication operator is implemented by this method.

Code	Result
`10 3 *`	30
`22.7 1.5 *`	34.05
`22.7 3/2 *`	34.05
`3/2 4/5 *`	6/5
`2+3i 3+4i *`	-6+17i

[a_numeric a_numeric] ** [a_numeric]

Routing: NOS

The exponentiation operator is implemented by this method. Note the the second operand, the power, is converted to a Float first.

Code	Result
`2 10 **`	1024
`3 4 **`	81
`2 1/2 **`	1.4142135623730951

[a_numeric a_numeric] + [a_numeric]

Routing: NOS

The addition operator is implemented by this method.

Code	Result
`1 1 +`	2
`1.0 7.2 +`	8.2
`1/2 1/3 +`	5/6
`1+2i 1+3i +`	2+5i

[a_numeric a_numeric] - [a_numeric]

Routing: NOS

The subtraction operator is implemented by this method.

Code	Result
`1 1 -`	0
`1.0 7.2 -`	-6.2
`1/2 1/3 -`	1/6
`1+2i 1+3i -`	0-1i

[a_numeric] .1/x [a_numeric]

Routing: TOS

This method computes the value of 1/x for the numeric argument. Note that division by zero produces an error in most cases except for 0.0 which produces Infinity.

Code	Result
`2 .1/x`	0
`2.0 .1/x`	0.5
`0 .1/x`	E15: divided by 0
`0.0 .1/x`	Infinity

[a_numeric] .10** [a_float]

Routing: TOS

This method computes 10^x for the numeric argument.

Code	Result
`3 .10**`	1000.0
`1/3 .10**`	2.154434690031884
`-3 .10**`	0.001

[a_numeric] .2** [a_float]

Routing: TOS

This method computes 2^X for the numeric argument.

Code	Result
`10 .2**`	1024.0
`-3 .2**`	0.125
`1+1i .2**`	1.5384778027279442+1.2779225526272695i

[a_numeric] .abs [a_numeric]

Routing: TOS

This method computes the absolute value of the argument number. Note that for complex numbers, this is the magnitude of the argument.

Code	Result
`1 .abs`	1
`1.0 .abs`	1.0
`1/1 .abs`	1/1
`-1 .abs`	1
`1+1i .abs`	1.4142135623730951

[a_numeric] .acos [a_float]

Routing: TOS

This method computes the arc-cosine $(\cos(x)^{-1})$ of the value. That is it computes the angle in radians whose cosine is the argument[78].

Code	Result
`1 .acos`	0.0
`0 .acos`	1.5707963267948966

78 Please see http://en.wikipedia.org/wiki/Inverse_trigonometric_functions for more details on inverse trigonometric functions.

[a_numeric] .acosh [a_float]

Routing: TOS

This method computes the arc-hyperbolic-cosine ($\cosh(x)^{-1}$) of the value. That is it computes the angle in radians whose hyperbolic-cosine is the argument[79].

Code	Result
`1 .acosh`	0.0

[a_numeric] .angle [a_float]

Routing: TOS

For complex numbers, this method computes the phase angle in radians of the number. For non complex numbers, this angle is zero for positive values and pi for negative ones.

Code	Result
`1+1i .angle`	0.7853981633974483
`1 .angle`	0.0
`-1 .angle`	3.141592653589793

[a_numeric] .asin [a_float]

Routing: TOS

This method computes the arc-sine ($\sin(x)^{-1}$) of the value. That is it computes the angle in radians whose sine is the argument.

Code	Result
`1 .asin`	1.5707963267948966
`0 .asin`	0.0

79 Please see http://en.wikipedia.org/wiki/Inverse_hyperbolic_function for more details on inverse hyperbolic trigonometric functions.

[a_numeric] .asinh [a_float]

Routing: TOS

This method computes the arc-hyperbolic-sine ($sinh(x)^{-1}$) of the value. That is it computes the angle in radians whose hyperbolic-sine is the argument

Code	Result
`1 .asinh`	0.881373587019543

[a_numeric] .atan [a_float]

Routing: TOS

This method computes the arc-tangent ($tan(x)^{-1}$) of the value. That is it computes the angle in radians whose tangent is the argument.

Code	Result
`1 .atan`	0.7853981633974483

[a_numeric a_numeric] .atan2 [a_float]

Routing: TOS

This is the two argument version of atan (arc-tangent).

For any real number arguments x, y not both equal to zero, atan2(y, x) is the angle in radians between the positive x-axis of a plane and the point given by the coordinates (x, y) on it. The angle is positive for counter-clockwise angles (upper half-plane, y > 0), and negative for clockwise angles (lower half-plane, y < 0)[80].

Code	Result

80 From http://en.wikipedia.org/wiki/Atan2.

[a_numeric] .atanh [a_float]

Routing: TOS

This method computes the arc-hyperbolic-tangent ($\tanh(x)^{-1}$) of the value. That is it computes the angle in radians whose hyperbolic-tangent is the argument

Code	Result
`0 .atanh`	0.0
`1 .atanh`	Infinity

[a_numeric a_numeric] .c2p [a_float a_float]

Routing: TOS

This method converts a two dimensional Cartesian coordinate to its equivalent Polar coordinate. On input the arguments are x, y. On output, the results are magnitude, angle (in radians).

Code	Result
`1 1 .c2p`	1.4142135623730951, 0.7853981633974483

[a_numeric] .cbrt [a_float]

Routing: TOS

This method computes the cube root ($X^{1/3}$) of the value.

Code	Result
`8 .cbrt`	2.0
`-8 .cbrt`	-2.0

[a_numeric] .ceil [an_integer]

Routing: TOS

This method computes the smallest integer that is greater than or equal to the argument.

Code	Result
`55.5 .ceil`	56
`-55.5 .ceil`	55

[a_numeric] .conjugate [a_numeric]

Routing: TOS

This method computes the complex conjugate of the argument. For non-complex data, this has no effect.

Code	Result
`2+3i .conjugate`	2-3i
`42 .conjugate`	42

[a_numeric] .cos [a_float]

Routing: TOS

This method computes the cosine[81] of the argument angle in radians.

Code	Result
`0 .cos`	1.0
`pi .cos`	-1.0

[a_numeric] .cosh [a_float]

Routing: TOS

This method computes the hyperbolic-cosine[82] of the argument angle in radians.

Code	Result
`0 .cosh`	0.0

[a_numeric] .cube [a_numeric]

Routing: TOS

This method computes the cube (X^3) of the number.

Code	Result
`3 .cube`	27
`3.0 .cube`	27.0

81 See http://en.wikipedia.org/wiki/Trigonometric_functions for further information.
82 See http://en.wikipedia.org/wiki/Hyperbolic_function for further information.

[a_numeric] .d2r [a_float]

Routing: TOS

This method converts the argument number from degrees to radians.

Code	Result
`180 .d2r`	3.141592653589793

[a_numeric] .denominator [an_integer]

Routing: TOS

This method extracts the denominator from the rational argument. If the argument is a float or complex, it extracts the denominator of the rationalized equivalent of that number. If the argument is an integer, the denominator is always one.

Code	Result
`1/3 .denominator`	3
`2.5 .denominator`	2
`1.5+2.25i .denominator`	4
`7 .denominator`	1

[a_numeric] .e** [a_float]

Routing: TOS

This method computes the value of e raised to the power of the argument (e^X).

Code	Result
`1 .e**`	2.718281828459045
`10 .e**`	22026.465794806718

[a_numeric] .emit []

Routing: TOS

This method emits a character with the code of numeric argument.

Code	Result
`65 .emit`	(Prints an “A”)

[a_numeric] .floor [an_integer]

Routing: TOS

This method computes the largest integer that is less than or equal to the argument.

Code	Result
`2.3 .floor`	2
`-2.3 .floor`	-3

[a_numeric a_numeric] .hypot [a_float]

Routing: TOS

Given two lengths, this method computes the length of the hypotenuse.

Code	Result
`3 4 .hypot`	5.0

[a_numeric] .imaginary [a_numeric]

Routing: TOS

This method extracts the imaginary component of a complex number. For non-complex numbers this is always zero.

Code	Result
`1+2i .imaginary`	2
`1-2i .imaginary`	-2
`42 .imaginary`	0

[a_numeric] .ln [a_float]

Routing: TOS

This method computes the natural logarithm ($\log_e$) of the given value.

Code	Result
`1 .ln`	0.0
`e .ln`	1.0
`10 .ln`	2.302585092994046

[a_numeric] .log10 [a_float]

Routing: TOS

This method computes the base 10 logarithm ($\log_{10}$) of the given value.

Code	Result
`1 .log10`	0.0
`e .log10`	0.4342944819032518
`10 .log10`	1.0

[a_numeric] .log2 [a_float]

Routing: TOS

This method computes the base 2 logarithm ($\log_2$) of the given value.

Code	Result
`2 .log2`	1.0
`e .log2`	1.4426950408889634
`10 .log2`	3.321928094887362

[a_numeric] .magnitude [a_numeric]

Routing: TOS

This method computes the magnitude of a complex value. For non-complex values, it computes the absolute value of the argument.

Code	Result
`3+4i .magnitude`	5.0
`-3 .magnitude`	3

[a_numeric] .numerator [a_numeric]

Routing: TOS

This method extracts the numerator from the rational argument. If the argument is a float or complex, it extracts the numerator of the rationalized equivalent of that number. If the argument is an integer, the numerator is the same as the number.

Code	Result
`1/3 .numerator`	1
`2.5 .numerator`	5
`1.5+2.25i .numerator`	6+9i
`7 .numerator`	7

[a_numeric a_numeric] .p2c [a_float a_float]

Routing: TOS

This method converts a two dimensional Polar coordinate to its equivalent Cartesian coordinate. On input the arguments are magnitude, angle (in radians). On output, the results are x, y.

Code	Result
`1 pi .p2c`	1.2246063538223773e-16, -1.0
`1 0 .p2c`	0.0, 1.0

[a_numeric] .polar [a_float a_float]

Routing: TOS

This method converts a complex number to a magnitude and an angle (in radians). For non-complex data, the result is the absolute value of the number and zero radians for positive values and pi radians for negative values.

Code	Result
`1+1i .polar`	1.4142135623730951, 0.7853981633974483
`5 .polar`	5, 0
`-5 .polar`	5, 3.141592653589793

[a_numeric] .r2d [a_float]

Routing: TOS

Convert a angle value from radians to degrees.

Code	Result
`Pi .r2d`	180.0

[error_float a_numeric] .rationalize_to [a_rational]

Routing: TOS

Convert a numeric into the simplest rational with an error not greater than the error_float. See the Rational class for more details.

[a_numeric] .real [a_numeric]

Routing: TOS

This method returns the real component of a complex number. For non-complex number, it simply returns the number unchanged.

Code	Result
`4+2i .real`	4

[a_numeric] .round [an_integer]

Routing: TOS

This method rounds the argument number to the nearest integer.

Code	Result
`4.1 .round`	4
`4.5 .round`	5
`4.9 .round`	5
`-4.1 .round`	-4
`-4.5 .round`	-5
`-4.9 .round`	-5

[num_digits a_number] .round_to [a_float]

Routing: TOS

Round the number to the specified number of digits past the decimal point. Negative digits round to places before the decimal point.

Note: Complex data are not supported.

Code	Result
`2 pi .round_to`	3.14
`-2 12345.666 .round_to`	12300
`2 3.12345+4i .round_to`	F40: Cannot coerce a Complex instance to a Float instance

[a_numeric] .sin [a_float]

Routing: TOS

This method computes the sine of the argument angle in radians.

Code	Result
`0 .sin`	0.0
`pi 2/ .sin`	1.0

[a_numeric] .sinh [a_float]

Routing: TOS

This method computes the hyperbolic-sine of the argument angle in radians.

Code	Result
`0 .sinh`	0.0
`pi 2/ .sin`	2.3012989023072947

[a_numeric] .sleep []

Routing: TOS

This method will put the current thread to sleep for the specified number of seconds. See the Thread class for more details.

[a_numeric] .sqr [a_numeric]

Routing: TOS

This method computes the square of the argument value.

Code	Result
`4 .sqr`	16
`2/3 .sqr`	4/9
`1+1i .sqr`	0+2i

[a_numeric] .sqrt [a_float]

Routing: TOS

This method computes the square root of the argument value.

Code	Result
`16 .sqrt`	4.0
`4/9 .sqrt`	0.6666666666666666

[a_numeric] .tan [a_float]

Routing: TOS

This method computes the tangent of the argument angle in radians.

Code	Result
`0 .tan`	0.0
`pi 4 / .tan`	0.9999999999999999

[a_numeric] .tanh [a_float]

Routing: TOS

This method computes the hyperbolic-tangent of the argument angle in radians.

Code	Result
`0 .tanh`	0.0
`1.0E6 .tanh`	1.0

[numeric] .to_t [a_time]

Routing: TOS

Convert the number to a time object. This is a helper method for the Time class.

[numeric] .to_t! [a_time]

Routing: TOS

Convert the number to a time object. This is a helper method for the Time class.

[an_object] .to_n [a_numeric]

Routing: TOS

Try to convert the object into the appropriate type of Numeric value. If this is not possible, return nil instead. Contrast with .to_n! This is a helper method of the Object class.

Code	Result
`2 .to_n`	2
`2.0 .to_n`	2.0
`"2" .to_n`	2
`"2.0" .to_n`	2.0
`"1/2" .to_n`	1/2
`"1+2i" .to_n`	1+2i
`"apple" .to_n`	nil

[an_object] .to_n! [a_numeric]

Routing: TOS

Try to convert the object into the appropriate type of Numeric value. If this is not possible, raise an error. Contrast with .to_n This is a helper method of the Object class.

Code	Result
`2 .to_n!`	2
`2.0 .to_n!`	2.0
`"2" .to_n!`	2
`"2.0" .to_n!`	2.0
`"1/2" .to_n!`	1/2
`"1+2i" .to_n!`	1+2i
`"apple" .to_n!`	`Cannot convert a String instance to a Numeric instance`

[a_numeric a_numeric] / [a_numeric]

Routing: NOS

This method implements the division operator.

Code	Result
`3 4 /`	0
`3.0 4 /`	0.75
`1 0 /`	E15: divided by 0
`1.0 0 /`	Infinity
`2/3 3 /`	2/9

[a_numeric] 0< [a_boolean]

Routing: TOS

Is this number less than zero?

Code	Result
`3 0<`	false
`0 0<`	false
`-3 0<`	true

[a_numeric] 0<= [a_boolean]

Routing: TOS

Is this number less than or equal to zero?

Code	Result
`3 0<=`	false
`0 0<=`	true
`-3 0<=`	true

[a_numeric] 0<=> [1, 0, or -1]

Routing: TOS

Perform a “three outcome” comparison of the value with zero.

Code	Result
`3 0<=>`	1
`0 0<=>`	0
`-3 0<=>`	-1

[a_numeric] 0<> [a_boolean]

Routing: TOS

Is the number not equal to zero?

Code	Result
`3 0<>`	true
`0 0<>`	false
`-3 0<>`	true

[a_numeric] 0= [a_boolean]

Routing: TOS

Is the number equal to zero?

Code	Result
`3 0=`	true
`0 0=`	false
`-3 0=`	true

[a_numeric] 0> [a_boolean]

Routing: TOS

Is the number greater than zero?

Code	Result
`3 0=`	true
`0 0=`	false
`-3 0=`	false

[a_numeric] 0>= [a_boolean]

Routing: TOS

Is the number greater than or equal to zero?

Code	Result
`3 0=`	true
`0 0=`	true
`-3 0=`	false

[a_numeric] 1+ [a_numeric]

Routing: TOS

Add one to the number

Code	Result
`2 1+`	3
`1/3 1+`	4/3

[a_numeric] 1- [a_numeric]

Routing: TOS

Subtract ond from the number

Code	Result
`2 1-`	1
`1/3 1-`	-2/3

[a_numeric] 2* [a_numeric]

Routing: TOS

Multiply the number by two.

Code	Result
`2 2*`	4
`1/3 2*`	2/3

[a_numeric] 2+ [a_numeric]

Routing: TOS

Add two to the number

Code	Result
`2 2+`	4
`1/3 2+`	7/3

[a_numeric] 2- [a_numeric]

Routing: TOS

Subtract two from the number

Code	Result
`2 2-`	0
`1/3 2-`	-5/3

[a_numeric] 2/ [a_numeric]

Routing: TOS

Divide the number by two. Note that for integers, rounding down toward negative infinity is employed.

Code	Result
`2 2/`	1
`1/3 2/`	1/6
`3 2/`	1
`-3 2/`	-2
`3.0 2/`	1.5
`-3.0 2/`	-1.5

[a_numeric a_numeric] < [a_boolean]

Routing: NOS

Is the first number less than the second?

Code	Result
`3 0 <`	false
`0 0 <`	false
`-3 0 <`	true

[a_numeric a_numeric] <= [a_boolean]

Routing: NOS

Is the first number less than or equal to the second?

Code	Result
`3 0 <=`	false
`0 0 <=`	true
`-3 0 <=`	true

[a_numeric a_numeric] <=> [-1, 0, or 1]

Routing: NOS

Perform a “three outcome” comparison of the first value with the second value.

Code	Result
`3 0 <=>`	1
`0 0 <=>`	0
`-3 0 <=>`	-1

[a_numeric a_numeric] > [a_boolean]

Routing: NOS

Is the first number greater than the second number.

Code	Result
`3 0 >`	true
`0 0 >`	false
`-3 0 >`	false

[a_numeric a_numeric] >= [a_boolean]

Routing: NOS

Is the first number greater than or equal to the second number.

Code	Result
`3 0 >=`	true
`0 0 >=`	true
`-3 0 >=`	false

[a_numeric a_numeric] mod [a_numeric]

Routing: NOS

Compute the modulus (or remainder) of dividing the first number by the second.

Code	Result
`5 3 mod`	2
`5.0 3.0 mod`	2.0
`7/3 1/4 mod`	1/12
`20.5 6 mod`	2.5
`10 0 mod`	E15: divided by 0
`10.0 0.0 mod`	E15: divided by 0

[a_numeric] neg [a_numeric]

Routing: TOS

Compute zero minus the number.

Code	Result
`5 neg`	-5
`2.3 neg`	-2.3
`1/3 neg`	-1/3

Object

Inheritance: Object ← nil

```
Object Shared Methods =
&&              .is_class?      .to_i!          .with{{         max
)methods        .name           .to_n           <>              min
.               .strlen         .to_n!          =               nil<>
.class          .to_duration    .to_r           ^^              nil=
.clone          .to_duration!   .to_r!          distinct?       not
.clone_exclude  .to_f           .to_s           f"              ||
.copy           .to_f!          .to_x           format
.init           .to_i           .to_x!          identical?

Object Shared Stubs =
!          .append{{  .open{{    0<>       2*        <=        and       p"
)stubs     .create{{  .select{{  0=        2+        <=>       com       parse
*          .do{{      /          0>        2-        >         mod       parse!
**         .each{{    0<         0>=       2/        >=        neg       xor
+          .map{{     0<=        1+        <         >>        or
-          .new{{     0<=>       1-        <<        @         p!"
```

The Object class is the root of the class tree. The fOOrth Object class has no parent class. This is why it is depicted above as being derived from nil. All other classes inherit from the Object class and gain its methods and functionality.

Instance Methods

[object_a object_b] && [true or false]

Routing: NOS

Logical AND for the case where the first operand is true. This takes on the value of the second operand converted to a Boolean value.

Note: Other parts of this method are implemented in the NilClass and FalseClass classes.

Code	Result
`false false &&`	false
`false true &&`	false
`true false &&`	false
`true true &&`	true

[object] . []

Routing: TOS.

Print out the object on the console using the default formatting.

Code	Result
`42 .`	(Prints out the answer to life, the universe and everything.)

[an_object] .:: method_name … ; []

Routing: VM.

Start defining an exclusive (singleton in Ruby parlance) method on the receiver object. The form of this definition takes the form:

```
<an object> .:: <method name> <code goes here> ;
```

For information on how the name affects the type of method created, see the section Routing above.

Notes

- Copies and clones of the affected object retain any additional methods added to that object before it was copied or cloned. See Cloning Data above for more details.
- Not all objects support exclusive methods. If that is the case then an error occurs.

Code	Result

`Array .new .name .`
 	
`[ 3 ] val$: $vv`
`$vv .:: .name "Fred" ;`
`$vv .name .`
 	
`$vv .copy .name .`
	(Prints) Array instance
	
	(Prints) Fred
	
	(Prints) Fred

`5 .:: foo 10 ;`	F13: Exclusive methods not allowed for type: Fixnum

Local Methods:

[value] val: local_name []

Routing: Compiler Context.

This method defines a local value in the current method.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 val: limit`	(Creates a value named limit set to 10)

[value] var: local_name []

Routing: Compiler Context.

This method defines a local variable in the current method.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 var: limit`	(Creates a variable named limit set to 10)

[value] val@: @instance_name []

Routing: Compiler Context.

This method creates an instance value in the instance of the host class.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 val@: @limit`	(Creates an instance value named @limit set to 10)

[value] var@: @instance_name []

Routing: Compiler Context.

This method creates an instance value in the instance of the host class.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 var@: @limit`	(Creates an instance variable named @limit initially set to 10)

[] super []

Routing: Self.

Call the previously defined method for this object. This allows a method to do all the things that the parent method did with customization of the parameters and additional pre and post actions.

[] ... ; []

Routing: Compiler Context.

Close off the compilation and its context. The method definition is terminated by this trailing semi-colon. After this method, this and the above local methods are no longer available.

[an_object] .class [a_class]

Routing: TOS.
Get the class of the receiver object.

Code	Result
`1/2 .class`	Rational
`"apple" .class`	String
`Nil .class`	NilClass

[an_object] .clone [an_object']

Routing: TOS.

Create a clone of receiver object. This clone is accomplished be performing a deep copy of the object, and all of its instance data and any data referenced by that data. The deep copy process has loop and cyclic graph detection to avoid going off into an infinite recursion. This contrasts with the VirtualMachine word “clone” which combines the actions of “dup .clone”.

See Cloning Data above for more details.

Code	Result
`@title .clone ": "  @name <<`	(By cloning @title, the string is not mutated.)

[] .clone_exclude [an_array_of_exclusions]

Routing: TOS

This method is part of the fOOrth implementation of the full clone protocol. This method is seldom called directly, instead it is called as a result of a call to the clone or .clone methods.

The purpose of the .clone_exclude method is to specify those data members that are excluded from the cloning process. For instance variables this is an array of strings with the names of those variables.

It is expected that sub-classes of the Object class will override this method and return an exclusion list appropriate for their needs.

Code Definition Example	Result
`MyClass .: .clone_exclude [ "@foo" ] ;`	(The variable @foo will no be cloned.)

[an_object] .copy [an_object']

Routing: TOS

Create a copy of receiver object. This copy is accomplished be performing a shallow copy of the object, and all of its instance data but not any data referenced by that data. This contrasts with the VirtualMachine word “copy” which combines the actions of “dup .copy”.

See Cloning Data above for more details.

Code	Result
`@title .clone ": "  @name <<`	(By cloning @title, the string is not mutated.)

[unspecified an_object] .init [unspecified]

Routing: TOS

The ".init" method is never called directly. Instead, this method is called by the ".new" method of the Class class. Its purpose is to perform any needed setup on the object being created. Parameters to the ".new" appear as parameters to the ".init" method. It is expected that user defined classes will override the ".init" method default in Object which takes no action.

In a hierarchy of classes, access to earlier versions of the .init method is possible with the super method (see super, a local method of ".:" in Class and ".::" in Object).

Code Definition Example	**Result**
`MyClass .: .init val@: @name ;`	(Creates an initialization method that takes an argument as the initial value for the name. In use it would appear as the usage example.
Usage Example	
`"Peter Camilleri" MyClass .new`	(Creates an instance of the MyClass class with the @name value set to the string "Peter Camilleri")

[an_object] .is_class? [false]

Routing: TOS

Is this Object a Class? No! Returns false. See the version of this method in Class for a more positive slant on things.

Code	**Result**
`Object .is_class`	true
`42 .is_class`	false

[an_object] .name [a_string]

Routing: TOS

Get the name of the object. For Class objects, this is the name of the class. For other objects, this is the name of their class followed by “instance”. For Virtual Machine instances, the name of the VM is also appended.

Code	Result
`Object .name`	“Object”
`100 .name`	“Fixnum instance”
`vm .name`	“VirtualMachine instance <Main>”

[an_object] .strlen [integer]

Routing: TOS

If the object is a string, determine the length of a string, else determine the length of the string created when the object is converted to a string (see .to_s for further info).

Code	Result
`"ABCD" .strlen`	4
`100 .strlen`	3
`Object .strlen`	6

[an_object] .to_duration [a_duration]

Routing: TOS

A helper method for the Duration class. See that class for more details.

[an_object] .to_duration! [a_duration]

Routing: TOS

A helper method for the Duration class. See that class for more details.

[an_object] .to_f [a_float or nil]

Routing: TOS

Try to convert the object to a float. See the Float class for more details.

[an_object] .to_f! [a_float]

Routing: TOS

Try to convert the object to a float. See the Float class for more details.

[an_object] .to_i [an_integer or nil]

Routing: TOS

Try to convert the object to an integer. See the Integer class for more details.

[an_object] .to_i! [an_integer]

Routing: TOS

Try to convert the object to an integer. If this is not possible raise an error. Contrast with .to_i! See the Integer class for more details.

[an_object] .to_n [a_numeric]

Routing: TOS

Try to convert the object into the appropriate type of Numeric value. See the Numeric class for more details.

[an_object] .to_n! [a_numeric]

Routing: TOS

Try to convert the object into the appropriate type of Numeric value. See the Numeric class for more details.

[an_object] .to_r [a_rational or nil]

Routing: TOS

Try to convert the object into a Rational. See the Rational class for more details.

[an_object] .to_r! [a_rational]

Routing: TOS

Try to convert the object into a Rational. See the Rational class for more details.

[an_object] .to_s [a_string]

Routing: TOS

Convert the object to a string. See the String class for more details.

[an_object] .to_x [a_complex or nil]

Routing: TOS

Try to convert the object into a Complex. See the Complex class for more details.

[an_object] .to_x! [a_complex]

Routing: TOS

Try to convert the object into a Complex. See the Complex class for more details.

[an_object] .with{{ … }} [unspecified]

Routing: NOS (since the Procedure Literal is TOS)

This method is used to override the default value of “self” in a embedded procedure literal block of code. The argument object is used as the “self” for the duration of the block. This allows access to ~ methods and instance data. It can also be a handy short-form access to the object. See the section Self for more details.

For more information on the methods local to the embedded procedure, see the Procedure class.

Code	Result
`42 .with{ 0 5 do i self + . space loop }`	(Prints) 42 43 44 45 46

`Object .new .with{`
`  10 var@: @limit`
`  self val$: $count }`
	(Creates an instance of the Object class, adds the instance variable, @limit and sets the global value $count to it.)

[object_a object_b] <> [a_boolean]

Routing: NOS

Return true if object_a does not equal object_b, else return false.

Code	Result
`42 42 <>`	false
`42 43 <>`	true
`"5" 5 <>`	true
`[ "5" ] [ "5" ] <>`	false
`[ "5" ] [ "6" ] <>`	true

[object_a object_b] = [a_boolean]

Routing: NOS

Return true if object_a is equal to object_b, else return false.

Code	Result
`42 42 =`	true
`42 43 =`	false
`"5" 5 =`	false
`[ "5" ] [ "5" ] =`	true
`[ "5" ] [ "6" ] =`	false

[object_a object_b] ^^ [true or false]

Routing: NOS.

Logical, exclusive OR for the case where the first operand is true. This takes on the opposite of the value of the second operand converted to a Boolean value.

Note: Other parts of this method are implemented in the NilClass and FalseClass classes.

Code	Result
`false false ^^`	false
`false true ^^`	true
`true false ^^`	true
`true true ^^`	false

[object_a object_b] distinct? [true or false]

Routing: NOS.

Return true if the objects a and b have distinct identities or values, else return false

Code	Result
`4 5 distinct?`	true
`4 4 distinct?`	false
`"hi" dup distinct?`	false
`"hi" "ho" distinct?`	true
`"hi" "hi" distinct?`	true

[an_object] f"a string” [a_string]

Routing: NOS

Create a string version of the object using the embedded formatting string. This is a helper method, see the String and Time classes for more details.

[an_object a_string] format [a_string]

Routing: NOS

Create a string version of the object using the specified formatting string. This is a helper method, see the String and Time classes for more details.

[object_a object_b] identical? [true or false]

Routing: NOS.

Return true if the objects a and b have identical identities and values, else return false

Code	Result
`4 5 identical?`	false
`4 4 identical?`	true
`"hi" dup identical?`	true
`"hi" "ho" identical?`	false
`"hi" "hi" identical?`	false

[object_a object_b] max [either object_a or object_b]

Routing: NOS

Return the larger or object_a or object_b. The object_b parameter is coerced to the same type as object_b for the comparison only. If the objects are equal in value, then object_b is returned.

Code	Result
`4 5 max`	5
`4 "5" max`	“5”
`4 2 max`	4
`4 "apple" max`	`Cannot coerce a String instance to an Integer instance`

[object_a object_b] min [either object_a or object_b]

Routing: NOS

Return the smaller or object_a or object_b. The object_b parameter is coerced to the same type as object_b for the comparison only. If the objects are equal in value, then object_b is returned.

Code	Result
`4 5 min`	4
`4 "5" min`	4
`4 2 min`	2
`4 "apple" min`	`Cannot coerce a String instance to an Integer instance`

[an_object] nil<> [true]

Routing: TOS

Is this object not equal to nil for the case where the object is not equal to nil. Always true, see NilClass for the flip side of this method.

Code	Result
`nil nil<>`	false
`false nil<>`	true
`0 nil<>`	true
`"" nil<>`	true

[an_object] nil= [false]

Routing: TOS

Is this object equal to nil for the case where the object is not equal to nil. Always false, see NilClass for the flip side of this method.

Code	Result
`nil nil=`	true
`false nil=`	false
`0 nil=`	false
`"" nil=`	false

[an_object] not [false]

Routing: TOS

Return the logical opposite for object for the case where the object is true. Always false, see FalseClass and NilClass for the flip sides of this method.

Code	Result
`nil not`	true
`false not`	true
`true not`	false
`0 not`	false
`"" not`	false

[object_a object_b] || [true]

Routing: NOS.

Logical, inclusive OR for the case where the first operand is true. This is true.

Note: Other parts of this method are implemented in the NilClass and FalseClass classes.

Code	Result
`false false &&`	false
`false true &&`	true
`true false &&`	true
`true true &&`	true

[an_object])methods []

Routing: TOS.

Display a formatted listing of the methods defined to the given object. This method is very similar to the)methods method defined in Class, except for the labeling of exclusive methods.

```
>[ 3 ] val$: $vv

>$vv .:: .name "Fred" ;

>$vv .name .
Fred
>$vv )methods
Exclusive Methods =
.name

Array Shared Methods =
!         .+midlr  .-midlr  .left    .midlr   .right   <<
+         .+right  .-right  .length  .min     .shuffle @
.+left    .-left   .[]!     .max     .pp      .sort
.+mid     .-mid    .[]@     .mid     .reverse .strmax
```

OutStream

Inheritance: OutStream ← Object

```
OutStream Class Methods =
.append   .append{{ .create   .create{{

OutStream Shared Methods =
.       .cr     .space  ~       ~cr     ~space
.close  .emit   .spaces ~"      ~emit   ~spaces

OutStream Class Stubs =
.new
```

The OutStream class is used to support the writing of information to files in an accessible file system.

Class Methods

[file_name OutStream] .append [an_outstream]

Routing: TOS

This method opens the file of the given name for appending. It returns an OutStream instance that may be used for writing data to that file. If the file in question does not exist, it is created. The programmer is responsible for managing that instance at all times. A typical strategy is to use a local value to hold that instance.

Note: The programmer is also responsible for ensuring that the file object is finally closed or data may be lost.

Code	Result
`"test.txt" OutStream .append val: wf`	(Creates a local value wf with an OutStream instance.

[file_name OutStream] .append{{ … }} [unspecified]

Routing: NOS (since the Procedure Literal is TOS)

This is actually a method proxy for OutStream. This method opens the file of the given name for appending, it then executes the embedded procedure literal block (between {{ and }}) with self set to the opened file for the duration of the block. Finally it closes the file.

For more information on the methods local to the embedded procedure, see the Procedure class.

Code	Result
`"test.txt"  OutStream` `.append{ ~"Hello" ~cr }`	(Appends Hello to the file.)

[file_name OutStream] .create [an_outstream]

Routing: TOS

This method create a file of the given name for output. It returns an OutStream instance that may be used for writing data to that file. If the file in question exists, it is replaced. The programmer is responsible for managing that instance at all times. A typical strategy is to use a local value to hold that instance.

Note: The programmer is also responsible for ensuring that the file object is finally closed or data may be lost.

Code	Result
`"test.txt" OutStream .create val: wf`	(Creates a local value wf with an OutStream instance.

[file_name OutStream] .create{{ … }} [unspecified]

Routing: NOS (since the Procedure Literal is TOS)

This is actually a method proxy for OutStream. This method creates a file of the given name for appending (if the file in question exists, it is replaced), it then executes the embedded procedure literal block (between {{ and }}) with self set to the opened file for the duration of the block. Finally it closes the file.

For more information on the methods local to the embedded procedure, see the Procedure class.

Code	Result
`"test.txt" OutStream` `.create{ ~"Hello" ~cr }`	(Create file with Hello.)

Instance Methods

[an_object an_outstream] . []

Routing: TOS

Print out the object to the output stream using the default formatting.

Code	Result
`42 wf .`	(Writes “42” to the output)

[an_outstream] .close []

Routing: TOS

Close the output stream object. After this, the file will not accept further data.

Code	Result
`wf .close`	(Close the file.)
`"Hello" wf .`	IOError detected: closed stream (See told ya!)

[an_outstream] .cr []

Routing: TOS

Add a new-line character to the output stream.

Code	Result
`wf .cr`	(Adds a new-line character to the output)

[a_number an_outstream] .emit [after]

Routing: TOS

Emits the number as a character to the output stream.

Code	Result
`65 wf .emit`	(Adds a letter “A” to the output)

[an_outstream] .space []

Routing: TOS

Adds a space to the output stream.

Code	Result
`wf .space`	(Adds a space to the output)

[count an_outstream] .spaces []

Routing: TOS

Adds the specified number of spaces to the output stream.

Code	Result
`5 wf .spaces`	(Adds five spaces to the output)

[an_object] ~ []

Routing: Self

Print out the object to the output stream using the default formatting.

Code	Result
`42 ~`	(Writes “42” to the output)

[] ~" … " []

Routing: Self

Print out the embedded string to the output stream.

Code	Result
`~"Hello"`	(Writes “Hello” to the output)

[] ~cr []

Routing: Self

Add a new-line character to the output stream.

Code	Result
`~cr`	(Adds a new-line character to the output)

[a_number] ~emit []

Routing: Self

Emits the number as a character to the output stream.

Code	Result
`65 ~emit`	

[] ~space []

Routing: Self

Adds a space to the output stream.

Code	Result
`~space`	(Adds a space to the output)

[count] ~spaces []

Routing: Self

Adds the specified number of spaces to the output stream.

Code	Result
`5 ~spaces`	(Adds five spaces to the output)

Class Stubs

The following method is stubbed out as it is not supported by the OutStream class.

.new

Procedure

Inheritance: Procedure ← Object

```
Procedure Shared Methods =
.call         .call_vx      .call_x       .start_named
.call_v       .call_with    .start

Helper Methods =
{{
```

The procedure class is used to represent anonymous methods, not tied to either the virtual machine or any object. They are objects in and of themselves and can be passed as arguments to methods or returned as values from methods.

Procedure Literals

Procedure literals are supported by the virtual machine method “{{“ and a locally defined method “}}”. The general usage is:

```
{{ (procedure body) }}
```

To be clear on the semantics involved: execution of a procedure literal pushes an instance of a procedure object onto the stack.

A valid management strategy is to place these procedures into values. For example:

```
{{ dup * }} var$: $proc
```

Procedure Literal Methods

[] {{ []

Routing: VM

This method opens the definition of a procedure literal. After the opening {{, the code that makes up the body of the procedure should be found.

Code	Result
4 {{ dup + }} .call	8

Local Methods:

[] v [value or nil]

Routing: Compiler Context.

In those contexts with a defined value, this method retrieves that value. When no such value exists, nil is returned.

Code	Result
`[ 1 2 3 ] .map{{ v dup * }}`	[1 4 9]
`{{ v }} .call`	nil

[] x [index or nil]

Routing: Compiler Context.

In those contexts with a defined index, this method retrieves that index. When no such value exists, nil is returned.

Code	Result
`[ 1 2 3 ] .map{{ x dup * }}`	[0 1 4]
`{{ x }} .call`	nil

[] … }} [a_procedure]

Routing: Compiler Context.

This method closes off the procedure literal context and delivers the resulting procedure object to the stack.

Code	Result
`{{ dup + }} .name`	“Procedure instance”

Instance Methods

[unspecified a_procedure] .call [unspecified]

Routing: TOS

Call the code in the procedure. The current self value is used as the self value of the procedure.

Code	Result
`3 $proc (see above) .call`	6

[unspecified v a_procedure] .call_v [unspecified]

Routing: TOS

Call the code in the procedure. The current self value is used as the self value of the procedure. In addition, a value (shown as “v” above) is passed into the the procedure and is accessible as via the v method.

Code	Result
`4 {{ v dup + }} .call_v`	8

[unspecified v x a_procedure] .call_vx [unspecified]

Routing: TOS

Call the code in the procedure. The current self value is used as the self value of the procedure. In addition, two values (shown as “v” and “x” above) are passed into the the procedure and is accessible as via the v and x methods.

Code	Result
`5 4 {{ v x 2dup + }} .call_vx`	5 4 9

[unspecified owner a_procedure] .call_with [unspecified]

Routing: TOS

Call the code in the procedure with the self value of that procedure set to the owner object.

Code	Result
`4 {{ self dup + }} .call_with`	8

[unspecified v x a_procedure] .call_x [unspecified]

Routing: TOS

Call the code in the procedure. The current self value is used as the self value of the procedure. In addition, a value (shown as “x” above) is passed into the the procedure and is accessible as via the x method.

Code	Result
`4 {{ x dup + }} .call_x`	8

[unspecified a_procedure] .start [unspecified a_thread]

Routing: TOS

Start the procedure object in its own thread. Any additional data on the stack is copied to the stack of the new virtual machine created with the new thread instance.

Note: The caller is responsible for removing or otherwise dealing with the additional data.

Code	Result
`3 {{ $proc .call . }} .start`	#<Thread:0xXXXXXXX> (Prints out 6)

[unspecified a_string a_procedure] .start_named [unspecified a_thread]

Routing: TOS

Start the procedure object in its own named thread. Any additional data on the stack is copied to the stack of the new virtual machine created with the new thread instance.

Note: The caller is responsible for removing or otherwise dealing with the additional data. The thread name however is removed by the method.

Code	Result
`"Fred" {{ vm .vm_name . }} .start`	#<Thread:0xXXXXXXX> (Prints out Fred)

Queue

Inheritance: Queue ← Object

```
Queue Shared Methods =
.clear    .empty?    .length    .pend    .pop    .push
```

The queue class implements a data “pipeline” which permits data objects to be inserted into the queue and retrieved from the queue in the order they were inserted. Queues are especially useful in buffering data for use by a producer thread to a consumer thread.

Queue objects are created using the default implementation of the .new method in the Object class.

Instance Methods

[a_queue] .clear []

Routing: TOS

This method removes the data elements from the queue.

Code	Result
`@q .clear`	(The queue is cleared.)

[a_queue] .empty? [a_boolean]

Routing: TOS

Is this queue empty?

Code	Result
`@q .empty?`	true or false

[a_queue] .length [count]

Routing: TOS

How many data elements reside in the queue?

Code	Result
`@q .length`	Count

[a_queue] .pend [an_object]

Routing: TOS

Wait for a data element in the queue. This method is intended for use by a “consumer” thread and enables it to wait for data to process from a “producer” thread.

Warning: If this operation creates a deadlock this may result in a fatal error or a program lock-up.

Code	Result
`@q .pend`	an_object

[a_queue] .pop [an_object]

Routing: TOS

Get a data element from the queue.

Note: If the queue is empty when this method is invoked, an error is raised.

Code	Result
`@q .pop`	an_object
`@q .pop`	F31: Queue Underflow: .pop

[an_object a_queue] .push []

Routing: TOS

Add a data element to the queue.

Code	Result
`42 @q .push`	(The data 42 is added to the queue)

Rational

Inheritance: Rational ← Numeric ← Object

```
Rational Shared Methods =
.split

Helper Methods =
.to_r      .to_r!     rational    .rationalize_to
```

Rational numbers[83] are those numbers that may be represented as a/b where a and b are both integers. Since rational numbers are implemented with fOOrth integers, they are not subject to (most) sizing restrictions and can thus represent numbers large and small with no loss of precision. Rational numbers inherit most of their methods from the Numeric class.

Rational Literals

Rational literals are supported directly by the compiler. Any number with an embedded '/' is considered to be a rational number. The regular expression detecting potential rational numbers is:

```
/\d\/\d/
```

Some example values follow:

Literal[84]	Value[85]
`1/2`	1/2
`1.2/3`	2/5

Instance Methods

[a_rational] .split [numerator denominator]

Routing: TOS.

Split a rational number into its two component parts.

Code	Result
1/2 .split	1 2
3/4 .split	3 4

83 See http://en.wikipedia.org/wiki/Rational_number

84 No spaces are permitted within the literal.

85 The numerator may be an integer or a float, the denominator must be an integer. See the respective sections for more details on those types of literals.

[an_object] .to_r [a_rational or nil]

Routing: TOS

Try to convert the object into a Rational. If this is not possible, return nil. Contrast with .to_r! This is a helper method of the Object class.

Code	Result
`2 .to_r`	2/1
`2.5 .to_r`	5/2
`"2.5" .to_r`	5/2
`"5/2" .to_r`	5/2
`"apple" .to_r`	nil

[an_object] .to_r! [a_rational]

Routing: TOS

Try to convert the object into a Rational. If this is not possible, raise an error. Contrast with .to_r! This is a helper method of the Object class.

Code	Result
`2 .to_r!`	2/1
`2.5 .to_r!`	5/2
`"2.5" .to_r!`	5/2
`"5/2" .to_r!`	5/2
`"apple" .to_r!`	`Cannot convert a String instance to a Rational instance`

[numerator denominator] rational [a_rational]

Routing: VM

This is a helper method of the Virtual Machine class. Given a numerator and denominator, create a rational number. This method is quite flexible in accepting a wide variety of numeric input. If, for some reason, the conversion cannot be performed, nil is returned.

Code	Result
`3 4 rational`	3/4
`3.5 4 rational`	7/8
`4 3.5 rational`	8/7
`1+2i 3 rational`	1/3+2/3i
`3.1 4 rational`	31/40
`"apple" 3 rational`	nil

[numerator denominator] rational! [a_rational]

Routing: VM

This is a helper method of the Virtual Machine class. Given a numerator and denominator, create a rational number. This method is quite flexible in accepting a wide variety of numeric input. If, for some reason, the conversion cannot be performed, an error occurs.

Code	Result
`3 4 rational!`	3/4
`3.5 4 rational!`	7/8
`4 3.5 rational!`	8/7
`1+2i 3 rational!`	1/3+2/3i
`3.1 4 rational!`	31/40
`"apple" 3 rational!`	F40: Cannot coerce a String instance, Fixnum instance to a Rational

[error_float a_numeric] .rationalize_to [a_rational]

Routing: TOS

Convert a numeric into the simplest rational with an error not greater than the error_float. If this is not possible, an error is generated.

This is a helper method of the Numeric class.

Code	Result
`0.01 pi .rationalize_to`	22/7
`0.001 pi .rationalize_to`	201/64
`0.01 1234/55 .rationalize_to`	157/7
`0.001 1234/55 .rationalize_to`	875/39
`0.01 1+5i .rationalize_to`	F20: A Complex instance does not understand .rationalize_to (:_156).

Stack

Inheritance: Stack ← Object

```
Stack Shared Methods =
.clear   .empty?   .length   .peek   .pop    .push
```

The stack class implements a data “well” which permits data objects to be inserted into the stack and retrieved in the reverse of the order they were inserted. Stacks are *not* thread safe and should not be used to communicate between threads.

Stack objects are created using the default implementation of the .new method in the Object class.

Instance Methods

[a_stack] .clear []

Routing: TOS

Clear out the data elements of the stack.

Code	Result
`@s .clear`	(The stack is cleared)

[a_stack] .empty? [a_boolean]

Routing: TOS

description

Code	Result
`@s .empty?`	true or false

[a_stack] .length [count]

Routing: TOS

How many data elements are in this stack?

Code	Result
`@s .length`	count

[a_stack] .peek [an_object]

Routing: TOS

Peek at the top-of-stack data element without removing it. Note that if there is no element to peek at, an error is thrown.

Code	Result
`@s .peek`	an_object
`@s .peek`	F31: Stack Underflow: .peek

[a_stack] .pop [an_object]

Routing: TOS

Get the top-of-stack data element. Note that if there is no element to get, an error is thrown.

Code	Result
`@s .pop`	an_object
`@s .pop`	F31: Stack Underflow: .pop

[an_object a_stack] .push []

Routing: TOS

Push a data element onto the stack

Code	Result
`42 @s .push`	(Push 42 onto the stack)

String

Inheritance: String ← Object

```
String Shared Methods =
*           .-left      .each{{     .load       .right      .throw      <=
+           .-mid       .emit       .lstrip     .right?     .to_lower   <=>
."          .-midlr     .left       .mid        .rjust      .to_t       >
.+left      .-right     .left?      .mid?       .rstrip     .to_t!      >=
.+mid       .call       .length     .midlr      .shell      .to_upper
.+midlr     .cjust      .lines      .posn       .split      <
.+right     .contains?  .ljust      .reverse    .strip      <<

Helper Methods =
.to_s       "           format      format"
```

The String class provides a wide range of character and string manipulating capabilities.

String Literals

String literals are directly supported by the compiler. Any method with a " character in it contains an embedded string literal. See String Literals in The Syntax and Style of fOOrth above. The most basic string literal uses a virtual machine macro ", but all methods with embedded strings work in a similar manner. It is illustrated below.

```
"string contents go here"
```

Within the string literal, characters usually represent themselves, but there are exceptions to this called escape sequences. Escape sequences allow the string literal to contain characters that do not fit the normal rules. These are shown below:

Escape Sequence	Interpretation
\"	A single " character
\\	A single \ character
\[86]	The string is continued on next line.
\n	A newline character.
\xFF[87]	An 8 bit character value.
\uFFFF[88]	A 16 bit character value.

86 This is a backslash character followed by the end-of-line character(s).
87 The FF represents a two digit hexadecimal value.
88 The FFFF represents a four digit hexadecimal value.

Embedded String Literals

Any method ending with a double quote mark (") will contain an embedded string literal. In effect the string value contained therein becomes an argument of the method that contains it.

It is important to understand that in methods with embedded strings, that string is pushed onto the stack *before* the method is invoked. Thus the top-of-stack will always be that string literal.

Multi-line String Literals

In fOOrth, string literals can span multiple lines by use of the backslash (\) character. In order for this to work, the backslash character needs to be the last character on the line. The string literal then picks up on the next line at the first non-blank character. For example:

```
>)show

[]
>"4567\

[]
>"   666"

["4567666"]
```

Note first how leading spaces on the continuation line are removed. Also note how a " is added to the prompt to remind the user that a string is in the process of being entered.

Lazy String Literals

A special case exists where a string is started on a line, and neither terminated with a closing double quote mark or a line extension mark, backslash. In this case, the fOOrth language performs a lazy string termination, closing the string and removing any trailing blank characters. This is shown below:

```
>"Test

>.
Test
>)"ls
Gemfile       demo.rb         fOOrth.reek  license.txt  rdoc      t.txt
Gemfile.lock  docs            integration  pkg          reek.txt  test.foorth
README.md     fOOrth.gemspec  lib          rakefile.rb  sire.rb   tests
```

Format Strings

The string formatting facility is a direct transplant of the Ruby mechanisms for the same[89]. A format string is a string with optional text and zero or more format sequences. The data being formatted may be in one of two forms. Either a discrete object or an array[90] of objects may be formatted, however, a lone object may only be used if the formatting string contains no more

89 The documentation for this feature is largely taken from the Ruby1.9.3-p448 Core API Reference.
90 Ruby supports formatting data from hashes, but fOOrth does not.

than one format sequence. The structure of a format sequence is shown below with elements in brackets representing optional components.

```
%[flags][width][.precision]type
```

The type parameter is a single character that describes the data being formatted. There are three major groups of types: Integer, Float, and Other.

Integer Format Types

Type	Format Description
b	Convert argument as a binary number. Negative numbers will be displayed as a two's complement prefixed with '..1'.
B	Equivalent to 'b', but uses an uppercase 0B for prefix if the alternative format indicated by # is active.
d	Convert argument as a decimal number.
i	Identical to 'd'.
o	Convert argument as an octal number. Negative numbers will be displayed as a two's complement prefixed with '..7'.
u	Identical to 'd'.
x	Convert argument as a hexadecimal number. Negative numbers will be displayed as a two's complement prefixed with '..f' (representing an infinite string of leading 'ff's).
X	Equivalent to 'x', but uses uppercase letters.

Float Format Types

Type	Format Description
e	Convert floating point argument into exponential notation with one digit before the decimal point as [-]d.dddddde[+-]dd. The precision specifies the number of digits after the decimal point (defaulting to six).
E	Equivalent to 'e', but uses an uppercase E to indicate the exponent.
f	Convert floating point argument as [-]ddd.dddddd, where the precision specifies the number of digits after the decimal point.
g	Convert a floating point number using exponential form if the exponent is less than -4 or greater than or equal to the precision, or in dd.dddd form otherwise. The precision specifies the number of significant digits.
G	Equivalent to 'g', but use an uppercase 'E' in exponent form.
a	Convert floating point argument as [-]0xh.hhhhp[+-]dd, which is consisted from optional sign, "0x", fraction part as hexadecimal, "p", and exponential part as decimal.
A	Equivalent to 'a', but use uppercase 'X' and 'P'.

Other Format Types

Type	Format Description
c	Argument is the numeric code for a single character or a single character string itself.
p	Convert the argument to a string using the Ruby argument.inspect.
s	Argument is a string to be substituted. If the format sequence contains a precision, at most that many characters will be copied.
%	A percent sign itself will be displayed. No argument taken. That is %% displays as a single % sign.

Formatting Flags

The flags are zero or more optional characters that modify how the formatting is done. Flags tend to be specific to certain types.

Flag	Applies to:	Description
space	Integer or Float	Leave a space at the start of non-negative numbers. For 'o', 'x', 'X', 'b' and 'B', use a minus sign with absolute value for negative values.
(digit)$	All	Specifies the absolute argument number for this field. Absolute and relative argument numbers cannot be mixed in a format string.
#	BboxX aAeEfgG	Use an alternative format. For the conversions 'o', increase the precision until the first digit will be '0' if it is not formatted as complements. For the conversions 'x', 'X', 'b' and 'B' on non-zero, prefix the result with "0x", "0X", "0b" and "0B", respectively. For 'a', 'A', 'e', 'E', 'f', 'g', and 'G', force a decimal point to be added, even if no digits follow. For 'g' and 'G', do not remove trailing zeros.
+	Integer or Float	Add a leading plus sign to non-negative numbers. For 'o', 'x', 'X', 'b' and 'B', use a minus sign with absolute value for negative values.
-	All	Left-justify the result of this conversion.
0	Integer or Float	Pad with zeros, not spaces. For 'o', 'x', 'X', 'b' and 'B', radix-1 is used for negative numbers formatted as complements.
*	All	Use the next argument as the field width. If negative, left-justify the result. If the asterisk is followed by a number and a dollar sign, use the indicated argument as the width.

The width is an optional numeric value that specifies the minimum size of the formatting field. Finally the optional precision specification is used to specify the number of digits past the decimal point for floating point data.

Examples

The following illustrates a very few of the possible formatting options:

Code	Result
`1234 f"%10d"`	`"      1234"`
`1234 f"%-10d"`	`"1234      "`
`1234 f"%010d"`	`"0000001234"`
`1234 f"%x"`	`"4d2"`
`1234 f"%x"`	`"4D2"`
`1234 f"%#x"`	`"0x4d2"`
`1234 f"%#X"`	`"0X4D2"`
`12.34 f"%f"`	`"12.340000"`
`12.34 f"%.2f"`	`"12.34"`
`12.34 f"%6.2f"`	`" 12.34"`
`12.34e6 f"%g"`	`"1.234e+07"`
`12.34e6 f"%#g"`	`"1.23400e+07"`
`"Hello World" f"%s"`	`"Hello World"`
`"Hello World" f"%15s"`	`"    Hello World"`
`"Hello World" f"%15.5s"`	`"          Hello"`
`[ 5 100 ] f"%*d"`	`"  100"`
`[ 6 2 12.34 ] f"%*.*f"`	`" 12.34"`
`100 f"% 4d%%"`	`" 100%"`
`-100 f"% 4d%%"`	`"-100%"`

Instance Methods

[a_string count] * [a_string]

Routing: NOS

Create a string with the original string repeated count times

Note: The original string is *not* mutated by this operation.

Code	Result
`"*" 10 *`	`"**********"`
`"Knock Knock Penny " 3 *`	`"Knock Knock Penny Knock Knock Penny Knock Knock Penny "`

[a_string an_object] + [a_string]

Routing: NOS

Create a new string that is the concatenation of the original string and the object, converted to a string if needed.

Note: The original string is *not* mutated by this operation.

Code	Result
`"Hello " "World" +`	`"Hello World"`
`"Hello " 42 +`	`"Hello 42"`

[] ."a string" []

Routing: TOS

Print the embedded string.

Code	Result
`."Hello World"`	(Prints Hello World)

[width an_object a_string] .+left [a_string]

Routing: TOS

Replace width characters on the left part of the string with the object, converted to a string if needed.

Note: The original string is *not* mutated by this operation.

Code	Result
`3 "123" "abcdefg" .+left`	“123defg”
`2 4552 "XX is the answer" .+left`	“4552 is the answer”

[posn width an_object a_string] .+mid [a_string]

Routing: TOS

Replace width characters, starting at posn of the string with the object, converted to a string if needed.

Note: The original string is *not* mutated by this operation.

Code	Result
`3 2 "XXXX" "abcdefgh" .+mid`	“abcXXXXfgh”

[left_posn right_posn an_object a_string] .+midlr [a_string]

Routing: TOS

Replace the characters, starting at left_posn and ending at right_posn (counted from the right), of the string with the object, converted to a string if needed.

Note: The original string is *not* mutated by this operation.

Code	Result
`3 3 "XXXX" "abcdefgh" .+midlr`	“abcXXXXfgh”

[width an_object a_string] .+right [a_string]

Routing: TOS

Replace width characters on the right part of the string with the object, converted to a string if needed.

Note: The original string is *not* mutated by this operation.

Code	Result
`3 "123" "abcdefg" .+right`	“abcd123”

[width a_string] .-left [a_string]

Routing:

Remove the width characters from the left of the string.

Note: The original string is *not* mutated by this operation.

Code	Result
`3 "abcdefg" .-left`	“defg”

[posn width a_string] .-mid [a_string]

Routing: TOS

Remove width characters from the string starting at the specified position.

Note: The original string is *not* mutated by this operation.

Code	Result
`2 4 "abcdefg" .-mid`	“abg”

[left_posn right_posn a_string] .-midlr [a_string]

Routing: TOS

Delete the characters, starting at left_posn and ending at right_posn (counted from the right), of the string.

Note: The original string is *not* mutated by this operation.

Code	Result
`1 1 "abcdefg" .-midlr`	“ag”

[width a_string] .-right [a_string]

Routing: TOS

Delete width characters from the right of the string.

Note: The original string is *not* mutated by this operation.

Code	Result
`2 "abcdefg" .-right`	“abcde”

[a_string] .call [unspecified]

Routing: TOS

Execute the string as code.

Note: This method can be a source of security problems, especially if the string being executed contains user input.

Code	Result
`"2 7 +" .call`	9

[width a_string] .cjust [a_string]

Routing: TOS

Create a string with the given string centered in a field of the specified width.

Note: The original string is *not* mutated by this operation.

Code	Result
`10 "abcd" .cjust`	“ abcd ”

[sub_string a_string] .contains? [a_boolean]

Routing: TOS

Return true if a_string contains the sub_string, else return false.

Code	Result
`"bcd" "abcdefg" .contains?`	true
`"b3d" "abcdefg" .contains?`	false

[a_string] .each{{ … }} []

Routing: NOS (since the Procedure Literal is TOS)

This method is the string iterator. It processes each character in the string in turn, calling the embedded procedure literal block (between {{ and }}) with the character value (v) and index (x) of the current array item.

For more information on the methods local to the embedded procedure, see the Procedure class.

Code	Result
`"Hello" .each{ v x + . space }`	(Prints out H0 e1 l2 l3 o4)
`"Hello" .each{ v dup + . space }`	(Prints out HH ee ll ll oo)
`"Hello" .each{ x . space }`	(Prints out 0 1 2 3 4)

[a_string] .emit []

Routing: TOS

Print the first chacter of the string

Code	Result
`"abcd" .emit`	(Prints “a”)

[width a_string] .left [a_string]

Routing: TOS

This method returns the left most width characters from the string.

Note: The original string is *not* mutated by this operation.

Code	Result
`2 "abcdefg" .left`	“ab”

[sub_string a_string] .left? [a_boolean]

Routing: TOS

This method determines in the string starts with the sub-string.

Code	Result
`"abc" "abcdefg" .left?`	true
`"ab3" "abcdefg" .left?`	false

[a_string] .length [count]

Routing: TOS

How many characters are in this string? Note that this count is of characters and not bytes.

Code	Result
`"abc" .length`	3
`"ab\uFFFFc" .length`	4

[a_string] .lines [a_array]

Routing: TOS

This method takes a string with optional embedded line feeds and produces an array of strings broken at those line feeds.

Note: The array strings do *not* contain any line feed characters.

Code	Result
`"qwer" .lines`	["qwer"]
`"qwer\nasdf\nzxcv\n" .lines`	["qwer", "asdf", "zxcv"]
`"qwer\nasdf\nzxcv" .lines`	["qwer", "asdf", "zxcv"]

[width a_string] .ljust [a_string]

Routing: TOS

Create a string with the given string left justified in a field of the specified width.

Note: The original string is *not* mutated by this operation.

Code	Result
`10 "abcd" .ljust`	“abcd ”

[a_string] .load [unspecified]

Routing: TOS

This method loads the file with the name given in the string. If no file type is specified, a type of ".foorth" is used as the default. The file is interpreted as fOOrth source code.

Note: This command is similar to the command)load"name" except that it does not report or provide feedback to the console.

Code	Result
"docs/snippets/times_table" .load	(loads the file times_table.foorth)

[a_string] .lstrip [a_string]

Routing: TOS

This method strips of any leading spaces on the left of the string

Note: The original string is *not* mutated by this operation.

Code	Result
" abc " .lstrip	"abc "

[posn width a_string] .mid [a_string]

Routing: TOS

This method extracts width characters starting at the specified position in the string.

Note: The original string is *not* mutated by this operation.

Code	Result
`3 2 "abcdefg" .mid`	"de"

[posn sub_string a_string] .mid? [a_boolean]

Routing: TOS

Return true if a_string contains the sub_string at the indicated position. Otherwise return false.

Code	Result
`2 "cde" "abcdefgh" .mid?`	true
`3 "cde" "abcdefgh" .mid?`	false

[left_posn right_posn a_string] .midlr [a_string]

Routing: TOS

This method extracts width characters starting left_posn and ending at right_posn (counted from the end of the string) from the specified string.

Note: The original string is *not* mutated by this operation.

Code	Result
`2 2 "abcdefgh" .midlr`	“cdef”

[sub_string a_string] .posn [a_posn or nil]

Routing: TOS

This method returns the first position that sub_string occurs within the specified string. If the sub_string does *not* occur, then nil is returned.

Code	Result
`"cde" "abcdefgh" .posn`	2
`"cdx" "abcdefgh" .posn`	nil

[a_string] .reverse [a_string]

Routing: TOS

Create a new string with the characters reversed.

Note: The original string is *not* mutated by this operation.

Code	Result
`"Able was I ere I saw Elba" .reverse`	“ablE was I ere I saw elbA”
`"pup" .reverse`	“pup”
`"dog" .reverse`	“god”

[width a_string] .right [a_string]

Routing: TOS

Return the width number of characters at the end (right side) of the string.

Note: The original string is *not* mutated by this operation.

Code	Result
`3 "abcdefgh" .right`	“fgh”

[sub_string a_string] .right? [a_boolean]

Routing: TOS

Does the string end with the characters of sub_string?

Code	Result
`"fgh" "abcdefgh" .right?`	true
`"f4h" "abcdefgh" .right?`	false

[width a_string] .rjust [a_string]

Routing: TOS

Create a string with the given string right justified in a field of the specified width.

Note: The original string is *not* mutated by this operation.

Code	Result
`10 "abcd" .rjust`	“ abcd”

[a_string] .rstrip [a_string]

Routing: TOS

This method strips of any trailing spaces on the right of the string

Note: The original string is *not* mutated by this operation.

Code	Result
" abc " .rstrip	“ abc”

[a_string] .split [an_array]

Routing: TOS

Given a string, create an array of strings by splitting along spaces. Multiple spaces still create only one split.

Note: The original string is *not* mutated by this operation.

Code	Result
`"abc def 123" .split`	["abc", "def", "123"]
`"abc def  123" .split`	["abc", "def", "123"]
`"abcdef123" .split`	["abcdef123"]

[a_string] .strip [a_string]

Routing: TOS

Create a string with leading and trailing spaces removed.

Note: The original string is *not* mutated by this operation.

Code	Result
" abc " .strip	“abc”

[a_string] .throw []

Routing: TOS

Signal an exception.

Note: This method is similar to the Virtual Machine method throw".

Code	Result
"A35: Grundle Skew Error” .throw	(Throws an error A35)

[a_string] .to_lower [a_string]

Routing: TOS

Create a new string with all the characters converted to lower case.

Note: The original string is *not* mutated by this operation.

Code	Result
"AbCd" .to_lower	“abcd”

[an_object] .to_s [a_string]

Routing: TOS

Convert the object to a string. This is a helper method of the Object class.

Code	Result
`2 .to_s`	“2”
`2.5 .to_s`	“2.5”
`5/2 .to_s`	“5/2”
`"apple" .to_s`	“apple”

[string] .to_t [a_time]

Routing: TOS

Convert the string to a time object. This is a helper method for the Time class.

[string] .to_t! [a_time]

Routing: TOS

Convert the string to a time object. This is a helper method for the Time class.

[a_string] .to_upper [a_string]

Routing: TOS

Create a new string with all the characters converted to upper case.

Note: The original string is *not* mutated by this operation.

Code	Result
"AbCd" .to_upper	“ABCD”

[a_string a_string] < [a_boolean]

Routing: NOS

Is the first string less than the second one?

Code	Result
"B" "A" <	false
"B" "B" <	false
"A" "B" <	true

[a_string an_object] << [a_string]

Routing: NOS

Append the string representation of the object onto the first string

Note: The original string *is* mutated by this operation.

Code	Result
"ABC" "DEF" <<	"ABCDEF"
"ABC" 42 <<	"ABC42"

[a_string a_string] <= [a_boolean]

Routing: NOS

Is the first string less than or equal to the second one?

Code	Result
"B" "A" <=	false
"B" "B" <=	true
"A" "B" <=	true

[a_string a_string] <=> [1, 0, -1]

Routing: NOS

Perform a “three outcome” comparison of the first value with the second value.

Code	Result
"B" "A" <=>	1
"B" "B" <=>	0
"A" "B" <=>	-1

[a_string a_string] > [a_boolean]

Routing: NOS

Is the first string greater than the second one?

Code	Result
"B" "A" >	true
"B" "B" >	false
"A" "B" >	false

[a_string a_string] >= [a_boolean]

Routing: NOS

Is the first string greater than or equal to the second one?

Code	Result
"B" "A" >=	true
"B" "B" >=	true
"A" "B" >=	false

[an_object] f"a format string" [a_string]

Routing: NOS

Create a string version of the object using the embedded formatting string. The code:

```
f"format string"
```

is short for:

```
"format string" format
```

This shorter form is generally more convenient except in those cases where the format string must be computed or retrieved from storage. See Format Strings above for more details. This is a helper method of the Object class.

Code	Result
`1234 f"%X"`	“4D2”
`[ 23 45 ] f"%X %X"`	“17 2D”

[an_object a_format_string] format [a_string]

Routing: NOS

Create a string version of the object using the specified formatting string.

See Format Strings above for more details. This is a helper method of the Object class.

Code	Result
`1234 "%X" format`	“4D2”
`[ 23 45 ] "%X %X" format`	“17 2D”

Thread

Inheritance: Thread ← Object

```
Thread Class Methods =
.current .list    .main    .new{{

Thread Shared Methods =
.alive? .exit   .join   .status .vm

Helper Methods =
.sleep   .start    pause

Thread Class Stubs =
.new
```

The Thread class is largely used to facilitate the management of the threads in a multi-threaded application. It also provides methods to access the main thread and to retrieve the virtual machine of a given thread.

Class Methods

[Thread] .current [a_thread]

Routing: TOS

Get the current thread.

Code	Result
`Thread .current`	#<Thread:0xXXXXXXX>

[Thread] .list [an_array]

Routing: TOS

Get a array of all currently running threads.

Code	Result
`Thread .list`	[#<Thread:0xXXXXXXX run>]

[Thread] .main [a_thread]

Routing: TOS

Get the main (first) thread of this application.

Code	Result
`Thread .main`	#<Thread:0xXXXXXXX>

[a_string Thread] .new{{ … }} [a_thread]

Routing: NOS (since the Procedure Literal is TOS)

This method is used to create a named thread with the name coming from the string and the body specified in the embedded procedure literal block bound by {{ … }}. The thread is returned to the caller.

For more information on the methods local to the embedded procedure, see the Procedure class.

Code	Result
`"Fred" Thread .new{ 5 .sleep }`	#<Thread:0xXXXXXXX>

Instance Methods

[a_thread] .alive? [a_boolean]

Routing: TOS

Is the thread argument alive? Returns true if it else and false if not.

Code	Result
`Thread .current .alive?`	true

[a_thread] .exit []

Routing: TOS

Instruct the thread argument to exit.

Code	Result
`Thread .current .exit`	(The thread exits)

[a_thread] .join []

Routing: TOS

Wait for the specified thread to exit.

Code	Result
`"Fred" Thread .new{ 5 .sleep } .join`	(Waits five seconds for the thread to exit)

[a_numeric] .sleep []

Routing: TOS

This method will put the current thread to sleep for the specified number of seconds.

This is a helper of the Numeric class.

Code	Result
`1 .sleep`	(Sleep for one second)
`0.1 .sleep`	(Sleep for one tenth of a second)
`1/1000 .sleep`	(Sleep for one millisecond)

[unspecified a_procedure] .start [a_thread]

Routing: TOS

Start the procedure object in its own thread. Any additional data on the stack is copied to the stack of the new virtual machine created with the new thread instance.

This is a helper method of the Procedure class.

Code	Result
`3 {{ $proc .call . }} .start`	#<Thread:0x211f2a8> (Prints out 6)

[a_thread] .status [a_string]

Routing: TOS

Get the status of the specified thread as a string

Code	Result
`Thread .current .status`	“run”

[a_thread] .vm [a_virtual_machine]

Routing: TOS

Retrieve the virtual machine associated with the thread.

Code	Result
`{{ Thread .list . }} .start .vm`	#<XfOOrth::VirtualMachine:0x1bb6898>

[] pause []

Routing: VM

This method causes the current thread to pause briefly to allow a chance for other threads to run.

This is a helper method of the Virtual Machine.

Code	Result
`pause`	(The thread pauses briefly)

Time

Inheritance: Time ← Object

```
Time Class Methods =
p!"          p"           parse        parse!

Time Shared Methods =
+          .as_zone  .minute   .second   .to_t!    <=         f"
-          .day      .month    .time_s   .utc?     <=>        format
.as_local  .fraction .offset   .to_a     .year     >
.as_utc    .hour     .sec_frac .to_t     <         >=

Helper Methods =
.to_t     local_offset   now

Time Class Stubs =
.new
```

The Time class is used to represent values of dates, times, and fractions of seconds. Times are represented internally as a number of seconds (and possibly fractions of seconds) since the initial time value (January 1, 1970 00:00 UTC). To be clear, the Time class is designed to operate in the current time frame. Attempting to use it for historical date references will likely yield incorrect results[91].

Creating Time Values

The Time class does not have a literal representation. In addition, the .new method is not supported. Time values are created by one of two strategies:

1. Using a special helper method, now, to create a time object for the present. See the now method in the section Special Time Values below.
2. Converting other values into time values with the ".to_t" and ".to_t!" methods. Both of these methods convert their argument value into a time value. If this is not possible, the "to_t" method returns nil while the "to_t!" method generates an error. The supported forms of conversion are listed below:

Source Type:	A Numeric[92]
Example:	`1434322201 .to_t`
Value:	2015-06-14 18:50:01 -0400
Description:	The numeric value describes the number of seconds (and fractions of seconds) since the initial time (January 1, 1970 00:00 UTC).

91 In addition the Time class only *represents* time values. In order to change the behavior of time, the optional Tardis attachment (not detailed in this document) is required.

92 Excluding Complex numeric values which generate an error.

Source Type:	A String
Examples:	`"Oct 26 1985 1:22" .to_t` `"Oct 26 1985 1:22 UTC" .to_t`
Values:	1985-10-26 01:22:00 -0400 1985-10-26 01:22:00 UTC
Description:	The string is converted to a time value using best-guess, sensible default assumptions. While it is possible to specify a time-zone, this can be very tricky in practice. If no time zone is specified, the machine local time zone is used. If UTC is specified, the Coordinated Universal Time[93] is used.
Note:	For more control over the conversion process from string to time, consider the parse methods below.

Source Type:	An Array
Example:	`[ 2015 6 14 18 50 0.0 -14400 ] .to_t` `[ 2015 6 14 18 ] .to_t`
Value:	2015-06-14 18:50:00 -0400 2015-06-14 00:00:00 -0400
Description:	The array contains values for the time components: year, month, day, hour, minutes, seconds, and offset from UTC. The array need not contain all of the values, missing data default to sensible values.

Note: It is also possible to convert a time value into a time value. This performs no action.

Special Time Values

[] local_offset [an_integer]

Routing: VM

This method returns the offset in seconds between local time and UTC. This value is not actually a constant as it is subject to change with factors such as daylight savings time and other manipulations of local time.

Code	Result
`local_offset`	-14400

93 UTC does indeed stand for Coordinated Universal Time. I hear that a committee was involved.

[] now [a_time]

Routing: VM

This method returns the current local time of the host computer system.

Code	Result
`now`	2015-06-17 11:51:30 -0400

Time Formatting

The standard[94] for formatting time values to strings is largely based on the strftime() function defined in ISO C[95] and POSIX[96]. Note that the same formatting codes are used by fOOrth for controlling the conversion of time objects into strings in the format and f" methods and the parsing of strings into time objects with the parse, parse!, p", and p!" methods.

These format codes, grouped in related categories are listed below:

Date formats (Year, Month, Day):	
Format	**Description**
%Y	Year with century if provided, will pad result at least 4 digits.
%C	The century (Year/100)
%m	Month of the year, zero-padded (01..12)
%_m	Month of the year, blank-padded (1..12)
%-m	Month of the year, with no-padding (1..12)
%B	The full month name (“January”)
%^B	The full month name in upper-case (“JANUARY”)
%b	The abbreviated month name (“Jan”)
%^b	The abbreviated month name in upper-case (“JAN”)
%h	Equivalent to %b
%d	Day of the month, zero-padded (01..31)
%-d	Day of the month, with no padding (1..31)
%e	Day of the month, blank-padded (1..31)
%j	Day of the year (001..366)

94 This material is largely taken from http://ruby-doc.org/core-2.2.0/Time.html#method-i-strftime
95 Please see: https://en.wikipedia.org/wiki/ANSI_C
96 Please see: https://en.wikipedia.org/wiki/POSIX

Time formats (Hour, Minute, Second, Subsecond):	
Format	**Description**
%H	Hour of the day, 24-hour clock, zero-padded (00..23)
%k	Hour of the day, 24-hour clock, blank-padded (0..23)
%I	Hour of the day, 12-hour clock, zero-padded (01..12)
%l[97]	Hour of the day, 12-hour clock, blank-padded (1..12)
%P	Meridian indicator, lowercase (``am'' or ``pm'')
%p	Meridian indicator, uppercase (``AM'' or ``PM'')
%M	Minute of the hour (00..59)
%S	Second of the minute (00..60)
%L	Millisecond of the second (000..999)
%N	Fractional seconds digits, default is 9 digits (nanosecond)
%3N	Fractional seconds digits, 3 digits (millisecond)
%6N	Fractional seconds digits, 6 digits (microsecond)
%9N	Fractional seconds digits, 9 digits (nanosecond)

Time zone formats:	
Format	**Description**
%z	Time zone as hour and minute offset from UTC (e.g. +0900)
%:z	Time zone as hour and minute offset from UTC with a colon (e.g. +09:00)
%::z	Time zone as hour, minute and second offset from UTC with a colon (e.g. +09:00:00)
%Z	Abbreviated time zone name or similar information. (OS dependent)

Weekday formats:	
Format	**Description**
%A	The full weekday name (“Sunday”)
%^A	The full weekday name in upper-case (“Sunday”)
%a	The abbreviated name (“Sun”)
%^a	The abbreviated name in upper-case (“Sun”)
%u	Day of the week (Monday is 1, 1..7)
%w	Day of the week (Sunday is 0, 0..6)

97 This character is a lower case “L”, and not an upper-case “I”.

ISO 8601 week-based year and week number formats:	
Note: The first week of YYYY starts with a Monday and includes YYYY-01-04. The days in the year before the first week are in the last week of the previous year.	
Format	**Description**
%G	The week-based year
%g	The last 2 digits of the week-based year (00..99)
%V	Week number of the week-based year (01..53)

Week number formats:	
Note: The first week of YYYY that starts with a Sunday or Monday (according to %U or %W). The days in the year before the first week are in week 0.	
Format	**Description**
%U	Week number of the year. The week starts with Sunday. (00..53)
%W	Week number of the year. The week starts with Monday. (00..53)

Seconds since the Epoch formats:	
Format	**Description**
%s	Number of seconds since 1970-01-01 00:00:00 UTC.

Literal strings:	
Format	**Description**
%n	A newline character (\n)
%t	A tab character (\t)
%%	A literal ``%'' character

Combination formats:	
Format	**Description**
%c	Date and time (%a %b %e %T %Y)
%D	Date (%m/%d/%y)
%F	The ISO 8601 date format (%Y-%m-%d)
%v	VAX[98] VMS date (%e-%^b-%4Y)
%x	Same as %D
%X	Same as %T
%r	12-hour time (%I:%M:%S %p)
%R	24-hour time (%H:%M)
%T	24-hour time (%H:%M:%S)

Class Methods

[a_string Time] p!" … " [a_time]

Routing: NOS

This is the form of the parse method with an embedded format string. For details on the parse string, see the section Time Formatting above.

If the source string cannot be parsed into a time, an error occurs. Contrast this with the p" method.

Code	Result
`"Sunday June 14 at 06:50 PM" Time p"%A %B %d at %I:%M %p"`	2015-06-14 18:50:00 -0400
`"Someday June 14 at 06:50 PM" Time p"%A %B %d at %I:%M %p!"`	F40: Cannot parse "Someday June 14 at 06:50 PM" into a Time instance

98 Oh such memories of the beloved (but *really* slow) VAX 11/780 of my college days.

[a_string Time] p" … " [a_time]

Routing: NOS

This is the form of the parse method with an embedded format string. For details on the parse string, see the section Time Formatting above.

If the source string cannot be parsed into a time, the value nil is returned. Contrast this with the p!" method.

Code	Result
`"Sunday June 14 at 06:50 PM" Time p"%A %B %d at %I:%M %p"`	2015-06-14 18:50:00 -0400
`"Someday June 14 at 06:50 PM" Time p"%A %B %d at %I:%M %p"`	nil

[a_string Time format_string] parse [a_time]

Routing: NOS

This is the form of the parse method with a format string. For details on the parse string, see the section Time Formatting above.

If the source string cannot be parsed into a time, the value nil is returned. Contrast this with the parse! method.

Code	Result
`"Sunday June 14 at 06:50 PM" Time "%A %B %d at %I:%M %p" parse`	2015-06-14 18:50:00 -0400
`"Someday June 14 at 06:50 PM" Time "%A %B %d at %I:%M %p" parse`	nil

a_string Time format_string] parse! [a_time]

Routing: NOS

This is the form of the parse method with a format string. For details on the parse string, see the section Time Formatting above.

If the source string cannot be parsed into a time, an error occurs. Contrast this with the parse method.

Code	Result
`"Sunday June 14 at 06:50 PM" Time "%A %B %d at %I:%M %p" parse!`	2015-06-14 18:50:00 -0400
`"Someday June 14 at 06:50 PM" Time "%A %B %d at %I:%M %p" parse!`	F40: Cannot parse "Someday June 14 at 06:50 PM" into a Time instance

Instance Methods

[time numeric] + [time]

Routing: NOS

Add the number of seconds in the numeric to the time to create a new time object in the future (or in the past if a negative number is added).

Note: The original time object is not mutated by this method.

Code	Result
`now 10 +`	2015-06-17 12:58:18 -0400 (A time 10 seconds in the future)

[time time or a_number] – [a_duration or a_time]

Routing: NOS

The time subtraction method has two distinct behaviors:

- In the first form, one time is subtracted from another, the result is a Duration with a span equal to the number of seconds between the two time values. The span is negative if the first time value is less than the second time value.
- In the second form, a number of seconds is subtracted from a time and the result is a new time in the past (or in the future if a negative number is subtracted.).

Note: The original time object is not mutated by this method.

Code	Result
`"1955-11-5 13:00" .to_t` `"Oct 26 1985 1:22" .to_t -`	Duration instance <-945865320.0 seconds>[99].
`now 10 -`	2015-06-17 13:30:21 -0400 (A time 10 seconds ago)

99 The approximate amount of time, in seconds, that Marty McFly travels on his first time travel voyage.

[time] .as_local [time]

Routing: TOS

Convert the given time to the same time in the local time zone. If the given time is already in the local time zone, then no change is made.

Note: The original time object is not mutated by this method.

Code	Result
`"12:00 UTC" .to_t .as_local`	2015-06-17 08:00:00 -0400

[time] .as_utc [time]

Routing: TOS

Convert the given time to the same time zone as UTC. If the given time is already UTC, then no change is made.

Note: The original time object is not mutated by this method.

Code	Result
`"8:00" .to_t .as_utc`	2015-06-17 12:00:00 +0000

[new_offset time] .as_zone [time]

Routing: TOS

Convert the given to to the time zone with the specified offset from UTC. If the given time already has that offset, no change is made.

Note:

- If the new offset used is local_offset, then this is the same as .as_local
- If the new offset used is 0, the this is the same as .as_utc.
- The original time object is not mutated by this method.

Code	Result
`3600 "12:00" .to_t .as_zone`	2015-06-17 17:00:00 +0100
`0 "8:00" .to_t .as_zone`	2015-06-17 12:00:00 +0000
`local_offset "12:00 UTC" .to_t .as_zone`	2015-06-17 08:00:00 -0400

[time] .day [day_of_month]

Routing: TOS

Extract the day of the month (1..31) from the time object.

Code	Result
`now .day`	17

[time] .fraction [float]

Routing: TOS

Extract the fractions of a second (0.0..0.9999...) from the time object.

Code	Result
`now .fraction`	0.725853

[time] .hour [integer]

Routing: TOS

Extract the hour (0..23) from the time object.

Code	Result
`now .hour`	14

[time] .minute [integer]

Routing: TOS

Extract the minute (0..59) from the time object.

Code	Result
`now .minute`	39

[time] .month [integer]

Routing: TOS

Extract the month (1..12) from the time object

Code	Result
`now .month`	6

[time] .offset [integer]

Routing: TOS

Extract from the time object, the offset in seconds from UTC.

Code	Result
`now .offset`	-14400

[time] .sec_frac [integer]

Routing: TOS

Extract the fractional seconds (0.0..59.9999...) from the time object.

Code	Result
`now .sec_frac`	32.578591

[time] .second [integer]

Routing:

Extract the whole seconds (0..59) from the time object.

Code	Result
`now .second`	24

[time] .time_s [string]

Routing: TOS

Convert the time to a reasonable string representation. For more control over the converion process, consider the format and f" methods instead.

Code	Result
`now .time_s`	Wed Jun 17 14:49:27 2015

[time] .to_a [array]

Routing: TOS

Break out all of the time information into an array of 7 elements. These elements, by position in the array, are:

```
[ Year Month Day Hour Minute Seconds Offset ]
```

These values are all integers except the Seconds which is a float value including any fractions of seconds.

Note: The array created by the .to_a method is compatible with the array required by the to_t and to_t! methods.

Code	Result
`now .to_a`	[2015 6 17 14 51 9.094222 -14400]

[array/numeric/string/time] .to_t [time]

Routing: TOS

Convert the argument into a time object. This method is actually a composite of three helper methods and a time method.

See Creating Time Values above for more details.

If it is not possible to convert the argument into a time object, nil is returned. Contrast with the to_t! method.

Code	Result
`[ 2015 6 17 14 51 9.09422 -14400 ] .to_t`	2015-06-17 14:51:09 -0400
`1434322201 .to_t`	2015-06-14 18:50:01 -0400
`"1955-11-5 13:00" .to_t`	1955-11-05 13:00:00 -0400
`now .to_t`	2015-06-17 15:18:33 -0400
`[ 2015 13 7 14 51 9.094222 -14400 ] .to_t`	nil
`infinity .to_t`	nil
`1+3i .to_t`	F20: A Complex instance does not understand .to_t (:_218).
`"apple" .to_t`	nil

[array/numeric/string/time] .to_t! [time]

Routing: TOS

Convert the argument into a time object. This method is actually a composite of three helper methods and a time method.

See Creating Time Values above for more details.

If it is not possible to convert the argument into a time object, an error occurs. Contrast with the to_t method.

Code	Result
`[ 2015 6 17 14 51 9.09422 -14400 ] .to_t!`	2015-06-17 14:51:09 -0400
`1434322201 .to_t!`	2015-06-14 18:50:01 -0400
`"1955-11-5 13:00" .to_t!`	1955-11-05 13:00:00 -0400
`now .to_t!`	2015-06-17 15:18:33 -0400
`[ 2015 13 7 14 51 9.09422 -14400 ] .to_t!`	F40: Cannot convert [2015, 13, 7, 14, 51, 9.09422, -14400] to a Time instance
`infinity .to_t!`	F40: Cannot convert Infinity to a Time instance
`1+3i .to_t!`	F20: A Complex instance does not understand .to_t (:_218).
`"apple" .to_t!`	F40: Cannot convert "apple" to a Time instance

[time] .utc? [a_boolean]

Routing: TOS

This method returns true if the argument is in the same time zone as UTC.

Code	Result
`now .utc?`	false
`"11:45 UTC" to_t .utc?`	true

[time] .year [integer]

Routing: TOS

Extract the year from the time object.

Code	Result
`now .year`	2015

[time time] < [a_boolean]

Routing: NOS

Is the first time less than the second one?

Code	Result
`1434322200 .to_t 1434322201 .to_t <`	true
`1434322201 .to_t 1434322201 .to_t <`	false
`1434322201 .to_t 1434322200 .to_t <`	false

[time time] <= [a_boolean]

Routing: NOS

Is the first time less than or equal to the second one?

Code	Result
`1434322200 .to_t 1434322201 .to_t <=`	true
`1434322201 .to_t 1434322201 .to_t <=`	true
`1434322201 .to_t 1434322200 .to_t <=`	false

[time time] <=> [1, 0, -1]

Routing: NOS

Perform a “three outcome” comparison of two time values.

Code	Result
`1434322200 .to_t 1434322201 .to_t <=>`	-1
`1434322201 .to_t 1434322201 .to_t <=>`	0
`1434322201 .to_t 1434322200 .to_t <=>`	1

[time time] > [a_boolean]

Routing: NOS

Is the first time greater than the second one?

Code	Result
`1434322200 .to_t 1434322201 .to_t >`	false
`1434322201 .to_t 1434322201 .to_t >`	false
`1434322201 .to_t 1434322200 .to_t >`	true

[before] >= [after]

Routing: NOS

Is the first time greater than or equal to the second one?

Code	Result
`1434322200 .to_t 1434322201 .to_t >=`	false
`1434322201 .to_t 1434322201 .to_t >=`	true
`1434322201 .to_t 1434322200 .to_t >=`	true

[time] f" … " [a_string]

Routing: NOS

Format the time value using the embedded format string as a template. See the section Time Formatting for more details on the available options.

Code	Result
`1434322201 .to_t f"%A %B %d at %I:%M %p"`	"Sunday June 14 at 06:50 PM"
`1434322201 .to_t f"%A %B %d, %r"`	"Sunday June 14, 06:50:01 PM"
`1434322201 .to_t f"%A %B %d, %T"`	"Sunday June 14, 18:50:01"

[time fmt_string] format [a_string]

Routing: NOS

Format the time value using the format string argument as a template. See the section Time Formatting for more details on the available options.

Code	Result
`1434322201 .to_t "%A %B %d at %I:%M %p" format`	"Sunday June 14 at 06:50 PM"
`1434322201 .to_t "%A %B %d, %r" format`	"Sunday June 14, 06:50:01 PM"
`1434322201 .to_t "%A %B %d, %T" format`	"Sunday June 14, 18:50:01"

Class Stubs

The following method is stubbed out in the Time class and not available: .new

TrueClass

Inheritance: TrueClass ← Object

```
No unique methods defined.

Helper Methods =
true
```

The TrueClass is a used to implement the true constant value. In spite of this, the functionality of true values is actually contained in the Object class.

TrueClass Literals

Instances of the TrueClass are made available by the Virtual Machine helper method “true”.

VirtualMachine

Inheritance: VirtualMachine ← Object

```
VirtualMachine Shared Methods =
!:            )restart     .start       an_hour      infinity     self
"             )show        .subclass:   begin        load"        space
)"            )start       .to_s        class:       local_offset spaces
)classes      )threads     .vm_name     clear        max_float    swap
)context      )time        2drop        clone        min_float    switch
)context!     )unmap"      2dup         complex      nan          throw"
)debug        )version     :            copy         nil          true
)elapsed      )vm          ?dup         cr           nip          try
)entries      )vm!         [            do           now          tuck
)globals      )words       a_day        dpr          over         val#:
)irb          -infinity    a_minute     drop         pause        val$:
)load"        .:           a_month      dup          pi           var#:
)map"         .::          a_second     e            pick         var$:
)nodebug      .dump        a_year       epsilon      rational     vm
)noshow       .elapsed     accept       false        rational!    {
)quit         .restart     accept"      if           rot          {{
```

The Virtual Machine is the center of activity of the fOOrth language system. It contains the stack, compiler, symbol mapping facility, context tracking and many other essential services utilized by the the entire class hierarchy. The same is also somewhat true of the Object class. The distinction between these is a matter of scope. The Object class is system wide. The virtual machine exists as a distinct instance per thread. Thus the virtual machine is better suited to managing the large scope of activities that are on a per-thread basis.

Since every thread must have exactly one virtual machine, it also serves as a universal routing target. Thus the virtual machine is used heavily by the compiler in the implementation of control and data literal structures.

The virtual machine is also the target for almost all user command level methods for the same reason.

Instance Methods

[] !: method_name … ; []

Routing: VM

This method is used to define a method on the virtual machine. These methods execute immediately even when in deferred or compile modes.

Notes

- There are pretty much no restrictions on the name except that it not contain spaces and any " signals an embedded string (which is also immediate).

Code	Result
`!: one 1 ;`	(Creates an immediate VM method one)

Local Methods:

[undefined] super [undefined]

Routing: Compiler Context.

Invoke the definition of this method in the nearest parent class of this class that defines a method of the same name. The arguments to the super method must match those of the parent class's implementation of this method. The return values will also be those of the parent class. If no parent class defines a method of the same name as this one, an error is generated.

Note: This method is rarely useful in Virtual Machine methods because so few of them override a parent method.

[value] val: local_name []

Routing: Compiler Context.

This method defines a local value in the current method.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 val: limit`	(Creates a value named limit set to 10)

[value] var: local_name []

Routing: Compiler Context.

This method defines a local variable in the current method.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 var: limit`	(Creates a variable named limit set to 10)

[value] val@: @instance_name []

Routing: Compiler Context.

This method creates an instance value in the instance of the host class.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 val@: @limit`	(Creates an instance value named @limit set to 10)

[value] var@: @instance_name []

Routing: Compiler Context.

This method creates an instance value in the instance of the host class.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 var@: @limit`	(Creates an instance variable named @limit initially set to 10)

[] ... ; []

Routing: Compiler Context.

Close off the method definition.

[] " … " [a_string]

Routing: VM

This is the string literal support method of the Virtual Machine. Please see class String Literals for more information.

[] -infinity [-Infinity]

Routing: VM

Please see Numeric – Special Numeric Values, above, for more information.

[a_class] .: method_name … ; []

Routing: VM

This method is used to define new methods on the specified class. Please see the Class class for more details.

[an_object] .:: []

Routing: VM.

Start defining an exclusive (singleton in Ruby parlance) method on the receiver object. Please see the Object class for more details.

[a_virtual_machine] .dump []

Routing: TOS

This debug method displays a “dump” of crucial data in the given virtual machine. See Tracking the Virtual Machine above for mere details.

Code	Result
`vm .dump`	(Displays a dump of the virtual machine)

[a_virtual_machine] .elapsed [a_float]

Routing: TOS

This method returns the number of seconds since the virtual machine was started or restarted.

Code	Result
`vm .elapsed`	68.952106 (Example only)

[a_class] .subclass: class_name []

Routing: VM

This method is used to create new classes. See the Class class for more details.

[a_virtual_machine] .restart []

Routing: TOS

Reset the start time of the virtual machine to now.

Code	Result
`vm .restart`	

[a_virtual_machine] .to_s [a_string]

Routing: TOS

Convert the virtual machine to a string. This method overrides the default implementation in the Object class.

Code	Result
`vm .to_s`	“VirtualMachine instance <Main>”

[a_virtual_machine] .start [a_date_time]

Routing: TOS

Get the start time of the virtual machine.

Code	Result
`vm .start`	a_time_object

[before] .vm_name [after]

Routing: VM

Return the name of the virtual machine as a string.

Code	Result
`vm .vm_name`	“Main”

[obj_a obj_b] 2drop []

Routing: VM

This method drops the top 2 elements from the data stack.

Code	Result
`1 2 3 4 2drop`	1 2

[obj_a obj_b] 2dup [obj_a obj_b obj_a obj_b]

Routing: VM

This method duplicates the top 2 elements on the stack.

Code	Result
`1 2 3 4 2dup`	1 2 3 4 3 4

[] : method_name … ; []

Routing: VM

This method is used to define a method on the virtual machine. These methods execute normally with to the current mode.

Notes

- There are pretty much no restrictions on the name except that it not contain spaces and any " signals an embedded string.
- This type of fOOrth method most closely resembles a classical FORTH word.

Code	Result
`: double dup + ;`	(Create the double method)

Local Methods:

[undefined] super [undefined]

Routing: Compiler Context.

Invoke the definition of this method in the nearest parent class of this class that defines a method of the same name. The arguments to the super method must match those of the parent class's implementation of this method. The return values will also be those of the parent class. If no parent class defines a method of the same name as this one, an error is generated.

Note: This method is rarely useful in Virtual Machine methods because so few of them override a parent method.

[value] val: local_name []

Routing: Compiler Context.

This method defines a local value in the current method.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 val: limit`	(Creates a value named limit set to 10)

[value] var: local_name []

Routing: Compiler Context.

This method defines a local variable in the current method.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 var: limit`	(Creates a variable named limit set to 10)

[value] val@: @instance_name []

Routing: Compiler Context.

This method creates an instance value in the instance of the host class.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 val@: @limit`	(Creates an instance value named @limit set to 10)

[value] var@: @instance_name []

Routing: Compiler Context.

This method creates an instance value in the instance of the host class.

See Data Storage in fOOrth, above, for more details on values and variables.

Code	Result
`10 var@: @limit`	(Creates an instance variable named @limit initially set to 10)

[] ... ; []

Routing: Compiler Context.

Close off the method definition.

[an_object] ?dup [an_object an_object] or [false/nil]

Routing: VM

Duplicate the top element of the stack unless it is false or nil.

Code	Result
`43 ?dup`	43 43
`true ?dup`	true true
`false ?dup`	false false
`nil ?dup`	nil nil

[] […] [an_array]

Routing: VM

This method is used to create array literal. See Array Literals above for more details.

[] a_day [a_duration]

Routing: VM

A helper method for Duration. See Special Duration Values for more information.

[] a_minute [a_duration]

Routing: VM

A helper method for Duration. See Special Duration Values for more information.

[] a_month [a_duration]

Routing: VM

A helper method for Duration. See Special Duration Values for more information.

[] a_second [a_duration]

Routing: VM

A helper method for Duration. See Special Duration Values for more information.

[] a_year [a_duration]

Routing: VM

A helper method for Duration. See Special Duration Values for more information.

[] accept [a_string]

Routing: VM

With a prompt of '? ', get a line of text from the console.

Code	Result
`accept`	a_string

[] accept" … " [a_string]

Routing: VM

Using the embedded string as a prompt, get a line of text from the console

Code	Result
`accept"Enter cost"`	a_string

[] an_hour [a_duration]

Routing: VM

A helper method for Duration. See Special Duration Values for more information.

[] begin … until/again/repeat []

Routing: VM

These methods are used to support arbitrary loops in fOOrth. There are three types of methods involved: begin, option while expression, and ending.

The begin method, marks the start of the loop.

The optional while method bails out of the loop if its argument is false. Zero or more while methods may be present in one loop.

Finally the ending methods until, again, or repeat, mark the end of the loop. The until method will exit the loop if given a true parameter. The again and repeat methods rely on a while method to terminate the loop.

This statement may be structured in many ways, here are a few examples:

```
(loop ends when condition is true)
begin (body) (condition) until

(loop end when condition is false)
begin (body) (condition) while (body) again

(loop end when condition1 or condition2 are false)
begin (body) (condition1) while (body) (condition2) while (body) again

(loop end when condition1 is false or condition2 is true)
begin (body) (condition1) while (body) (condition2) until
```

Where (body) represents the loop body code, and (condition) is a loop test or exit criteria. See the begin statement above for more details.

Local Methods:

[a_boolean] while []

Routing: Compiler Context.

This method exits the loop if the boolean is false.

[a_boolean] … until []

Routing: Compiler Context.

This method marks the end of the loop. Further, it ends the loop if the boolean is true.

[] ... again []

Routing: Compiler Context.

This method marks the end of the loop. This method is interchangeable with repeat.

[] … repeat []

Routing: Compiler Context.

This method marks the end of the loop. This method is interchangeable with again.

[] class: class_name []

Routing: VM

This method is used to create new classes. See the Class class for more details.

[an_object] clone [an_object an_object]

Routing: VM

Take the top element of the stack and create a clone (deep copy) of it. The original object becomes the second element on the stack.

Code	Result
`"apple" clone`	“apple” “apple”
`"apple" clone distinct?`	true

[real_part imaginary_part] complex [a_complex]

Routing: VM

Given two numbers, create a complex number. See the Complex class for more details.

[an_object] copy [an_object an_object]

Routing: VM

Take the top element of the stack and create a (shallow) copy of it. The original object becomes the second element on the stack.

Code	Result
`"apple" clone`	“apple” “apple”
`"apple" clone distinct?`	true

[] cr []

Routing: VM

Send a new line character to the console.

Code	Result
`cr`	(A new line is sent to the console.)

[start_value end_stop] do … loop/+loop []

Routing: VM

The do loop is used to facilitate counted iteration in fOOrth. In general, looping proceeds from the start_value up to the end_stop-1. Access to the iteration value is provided by the “i” method while “-i” supplies the reverse iteration value. The related “j” and “-j” methods allow access to an “outer” loop iteration value. Finally the “loop” method closes off the loop body and adds one to the iteration value, while “+loop” allows the iteration step to be specified.

Note: In order to work correctly the “-i”/”-j” methods require the end_stop to be one more than the last value iterated.

See the do statement above for more information.

Code	Result
`0 10 do i . space loop`	(Prints 0 1 2 3 4 5 6 7 8 9)
`0 10 do -i . space loop`	(Prints 9 8 7 6 5 4 3 2 1 0)
`0 9 do i . space 2 +loop`	(Prints 0 2 4 6 8)
`0 9 do -i . space 2 +loop`	(Prints 8 6 4 2 0)
`0 10 do -i . space 2 +loop`	(Prints 9 7 5 3 1)

Local Methods:

[] i [iteration_value]

Routing: Compiler Context.

Get the iteration value of the do loop. See above.

[] j [iteration_value]

Routing: Compiler Context.

Get the iteration value of an outer loop. This is to support nested loops. If there is no outer loop, the value zero is returned.

Code	Result

`0 2 do`
`  0 2 do i . ."," j . space loop`
`loop`
	(Prints 0,0 1,0 0,1 1,1)

`0 2 do j . space loop`	(Prints 0 0)

[] -i [iteration_value]

Routing: Compiler Context.

Get the reverse iteration value. Specifically, this is computed as:

```
(start_value + end_stop - 1) - i.
```

Code	Result
`10 20 do -i .  space loop`	(Prints 19 18 17 16 15 14 13 12 11 10)

[] -j [iteration_value]

Routing: Compiler Context.

Get the reverse iteration value of an outer loop. This supports reverse outer nested loops. If there is no outer loop, the value zero is returned.

Code	Result

`0 2 do`
`  0 2 do -i . ."," -j . space loop`
`loop`
	(Prints 1,1 0,1 1,0 0,0)

`0 2 do j . space loop`	(Prints 0 0)

[] ... loop []

Routing: Compiler Context.

This method marks the end of the loop.

[step_value] … +loop []

Routing: Compiler Context.

This method increments the loop index by the specified value and marks the end of the loop. If the step value is not a number or is less than or equal to zero, an error occurs.

Code	Result
`0 10 do i . space 2 +loop`	(Displays “0 2 4 6 8”)
`0 10 do i . space "apple" +loop`	F40: Cannot convert a String instance to a Numeric instance
`0 10 do i . space 0 +loop`	F41: Invalid loop increment value: 0

[] dpr [a_float]

Routing: VM

A helper method for Numeric. See Special Numeric Values for more information.

[an_object] drop []

Routing: VM

Drop the top-of-stack element. If there is no such element, an error occurs.

Code	Result
`1 2 3 drop`	1 2
`drop`	F30: Data Stack Underflow: pop

[an_object] dup [an_object an_object]

Routing: VM

Duplicate the top-of-stack element. This only duplicates references, not the data itself

Code	Result
`4 dup`	4 4
`"apple" dup`	“apple” “apple”
`"apple" dup identical?`	true
`"apple" dup distinct?`	false

[] e [a_float]

Routing: VM

A helper method for Numeric. See Special Numeric Values for more information.

[] epsilon [a_float]

Routing: VM

A helper method for Numeric. See Special Numeric Values for more information.

[] false [false]

Routing: VM

A helper method for FalseClass. See FalseClass literals for more information.

[a_boolean] if … then [unspecified]

Routing: VM

The “if” method implements the classic if statement in fOOrth. The if statement is used for cases where one or two branches are required. Where an arbitrary number of code branches, please see the switch statement below. The generalized layout of the if statement is:

```
(a condition) if (true clause) else (false clause) then
```

Where the else clause is optional, and only one of them is allowed. See the if statement above for more details.

Code	Result
`2 odd? if ."ODD" else ."EVEN" then`	(Prints EVEN)
`3 odd? if ."ODD" else ."EVEN" then`	(Prints ODD)
`true if space else cr else ."?" then`	F11: ?else?

Local Methods:

[] … else … []

Routing: Compiler Context.

This marks the beginning of the optional else clause.

[] ... then []

Routing: Compiler Context.

This marks the end of the if statement.

[] infinity [a_float]

Routing: VM

A helper method for Numeric. See Special Numeric Values for more information.

[] load"file_name" [unspecified]

Routing: VM

This methods loads a fOOrth source file, executing the code contained therein. If no file type is given, a type of ".foorth" is used as a default.

Note: This method is similar to the)load" command except that it does not provide feedback to the console.

Code	Result
`load"my_file.foorth"`	(Loads my_file.foorth)
`load"my_file"`	(Loads my_file.foorth)
`load"my_file."`	(Loads my_file.)

[] max_float [a_float]

Routing: VM

A helper method for Numeric. See Special Numeric Values for more information.

[] min_float [a_float]

Routing: VM

A helper method for Numeric. See Special Numeric Values for more information.

[] nan [a_float]

Routing: VM

A helper method for Numeric. See Special Numeric Values for more information.

[before] nil [nil]

Routing: VM

A helper method for NilClass. See NilClass Literals for more information.

[object_a object_b] nip [object_b]

Routing: VM

Drop the next-of-stack element. The top-of-stack element is not affected.

Code	Result
`1 2 3 nip`	1 3

[object_a object_b] over [object_a object_b object_a]

Routing: VM

Push the next-of-stack element onto the stack. This becomes the new top-of-stack element.

Code	Result
`1 2 3 over`	1 2 3 2

[] pause []

Routing: VM

Pause the current thread. See the Thread class above for more details.

[] pi [a_float]

Routing: VM

A helper method for Numeric. See Special Numeric Values for more information.

[index] pick [an_object]

Routing: VM

This method picks off the item on the stack selected by the index, where 1 represents the top-of-stack, 2 the next-of-stack, etc. Attempting to read elements deeper than the total stack or “before” the top-of-stack generate an error.

Code	Result
`1 2 3 1 pick`	1, 2, 3, 3
`1 2 3 3 pick`	1, 2, 3, 1
`1 2 3 "apple" pick`	F40: Cannot coerce a String instance to an Integer instance
`1 2 3 0 pick`	F30: Data Stack Underflow: peek

[numerator denominator] rational [a_rational]

Routing: VM

Convert a numerator and denominator into a rational number. See the Rational class for more details.

[numerator denominator] rational! [a_rational]

Routing: VM

Convert a numerator and denominator into a rational number. See the Rational class for more details.

[object_a object_b object_c] rot [object_b object_c object_a]

Routing: VM

This method “rotates” the top three elements on the stack.

Code	Result
`1 2 3 rot`	2 3 1

[] self [a_object]

Routing: VM

This method pushes the current “owner” object onto the stack.

See the section Self, above, for more details.

Code	Result
`>self`	VirtualMachine instance <Main>
`4 .with{ self }`	4

`class: MyClass`
`MyClass .: .who_r_u self ;`
`MyClass .new .who_r_u .name`
	“MyClass instance”

[] space []

Routing: VM

Prints a space character on the console.

Code	Result
`space`	(Prints a space)

[count] spaces []

Routing: VM

Prints count spaces on the console.

Code	Result
`1 . 5 spaces 1 .`	(Prints 1 1)

[object_a object_b] swap [object_b object_a]

Routing: VM

This method exchanges (swaps) the top two elements of the stack.

Code	Result
`1 2 swap`	2 1

[unspecified] switch … end [unspecified]

Routing: VM

The switch statement is used to create a program decision point with an arbitrary number of code branches. Recall that the if statement is used for cases where one or two branches suffice.

In general, the switch statement is bound by the key words “switch” and “end”. In between, arbitrary code is permitted with two additional local methods: “break” and “?break”. When a break is executed, the program “jumps” to the end of the switch and exits. The ?break is the same except that this jump action is only taken if its argument is not false or nil.

The switch statement is laid out along the following lines:

```
switch
  condition1 if action1 break then
  condition2 if action2 break then
  condition3 ?break
  condition4 if action4 break then

  (etc)

  default_action_here
end
```

There are many possible ways to utilize the switch statement, the above is merely one example. See the switch statement above for more details.

Local Methods:

[] break []

Routing: Compiler Context.

This method breaks out of the switch code block.

[a_boolean] ?break []

Routing: Compiler Context.

This method breaks out of the switch code block if the argument is true, else it takes no action.

[] … end []

Routing: Compiler Context.

This method marks the end of the switch block.

[] throw"Error Message" []

Routing: VM

This method is used to send exception messages. These messages consist of strings with a formal error code followed by a free-form descriptive message.

See Handling Exceptions above for more details.

Code	Result
`throw"U01: Invalid User Id."`	(Throws the exception U01)

[] true [true]

Routing: VM

A helper method for TrueClass. See TrueClass Literals for more information.

Code	Result
`true`	true

[unspecified] try … end [unspecified]

Routing: VM

The try block is used to control and contain exceptions. The try block is composed of three sections:

- The try section that contains the potentially error prone code.
- The optional catch section that responds to and processes exceptions.
- The optional finally section that performs clean-up actions regardless of whether any exceptional conditions were encountered.

A simple example try block is:

```
: ttry
  try swap dup . ." / " swap dup . ." = " / .
  catch
    switch
      ?"E15" if clear ."Error" break then
      bounce
    end
  finally
    cr ."Done!" cr
  end ;
```

```
>100 2 ttry
100 / 2 = 50
Done!

>10 0 ttry
10 / 0 = Error
Done!
```

See Handling Exceptions above for more details.

Local Methods:

[] catch []

Routing: Compiler Context.

This method starts the exception handling portion of the try block.

[] ?"error_code" [a_boolean]

Routing: Compiler Context.

This method matches the error code of the current exception with the embedded string. The strings are compared only as far as the length of the embedded string. If the sub-strings match, true is returned, else false. This method is only permitted in the catch section.

Code	**Result**
`?"E"`	(Matches all errors codes starting with “E”)
`?"E05"`	(Matches all “E05” codes, Index Error)
`?"E05,01"`	(Matches all “E05,01” codes, Key Error)

[] bounce []

Routing: Compiler Context.

Relaunch the current error so that some higher level catch clause can deal with it. This method is only permitted in the catch section.

[] error [a_string]

Routing: Compiler Context.

Retrieve the full text of the current error message. This method is only permitted in the catch section.

[] finally []

Routing: Compiler Context.

This method starts the cleanup section of the try block. The finally section is always executed regardless of caught or un-caught exceptions that may occur.

[before] … end [after]

Routing: Compiler Context.

The method closes off the try block.

[object_a object_b] tuck [object_b object_a object_b]

Routing: VM

This method tucks the top element of the stack under the second element.

Code	Result
`1 2 tuck`	2 1 2

[an_object] val#: thread_data_name []

Routing: VM

This method is used to define a thread local value. This value is available at all points in this thread. A copy of this value will be made in any threads created after this value is created. The name of the value created must conform to the following regex:

```
/^#[a-z][a-z0-9_]*$/
```

This means the the name must start with a "#" and a lower case letter followed by zero or more lower case letters or digits or underscores "_".

See Data Storage in fOOrth – Scoping, for more details on this topic.

Code	Result
`42 val#: #answer`	(Creates the thread value #answer)
`42 val#: wrong`	F10: Invalid val name wrong

[an_object] val$: global_data_name []

Routing: VM

This method is used to define a global value. This value is available to all points in the fOOrth program. The name of the value created must conform to the following regex:

```
/^\$[a-z][a-z0-9_]*$/
```

This means the the name must start with a “$” and a lower case letter followed by zero or more lower case letters or digits or underscores “_”.

See Data Storage in fOOrth – Scoping, for more details on this topic.

Code	Result
`42 val$: $answer`	(Creates the global value $answer)
`42 val$: wrong`	F10: Invalid val name wrong

[an_object] var#: thread_data_name []

Routing: VM

This method is used to define a thread local variable. This variable is available at all points in this thread. A copy of this variable will be made in any threads created after this variable is created. The name of the variable created must conform to the following regex:

```
/^#[a-z][a-z0-9_]*$/
```

This means the the name must start with a “#” and a lower case letter followed by zero or more lower case letters or digits or underscores “_”.

See Data Storage in fOOrth – Scoping, for more details on this topic.

Code	Result
`42 var#: #answer`	(Creates the thread variable #answer)
`42 var#: wrong`	F10: Invalid var name wrong

[an_object] var$: global_data_name []

Routing: VM

This method is used to define a global variable. This variable is available to all points in the fOOrth program. The name of the variable created must conform to the following regex:

```
/^\$[a-z][a-z0-9_]*$/
```

This means the the name must start with a “$” and a lower case letter followed by zero or more lower case letters or digits or underscores “_”.

See Data Storage in fOOrth – Scoping, for more details on this topic.

Code	Result
`42 var$: $answer`	(Creates the global variable $answer)
`42 var$: wrong`	F10: Invalid var name wrong

[] vm [a_virtual_machine]

Routing: VM

Get the current thread's virtual machine instance.

Code	Result
`vm .name`	“VirtualMachine instance <Main>”

[] { … } [a_hash]

Routing: VM

This is a helper method for creating Hash objects. See Hash Literals above for more details.

[] {{ … }} [a_procedure]

Routing: VM

This is a helper method for creating Procedure objects. See Procedure Literals above for more details.

[])" ... " []

Routing: VM

This command submits the embedded string as input to the console's command processor. Some examples of this command in action are:

```
>)"ls"
Gemfile       demo.rb         fOOrth.reek  license.txt  rdoc      t.txt
Gemfile.lock  docs            integration  pkg          reek.txt  test.foorth
README.md     fOOrth.gemspec  lib          rakefile.rb  sire.rb   tests

>)"git status"
On branch master
Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   docs/The_fOOrth_User_Guide.odt
        modified:   lib/fOOrth/compiler.rb
        modified:   lib/fOOrth/compiler/parser.rb

no changes added to commit (use "git add" and/or "git commit -a")
```

Note: This command can be used to do very naughty things to the host computer.

[])classes []

Routing: VM

Generate a list of the classes in the system. For example:

```
>)classes
Array           Fixnum          NilClass        Queue           TrueClass
Bignum          Float           Numeric         Rational        VirtualMachine
Class           Hash            Object          Stack
Complex         InStream        OutStream       String
FalseClass      Integer         Procedure       Thread
```

[])context []

Routing: VM

Display the compiler context at execution of the)context command. See the section Context for more details.

[])context! []

Routing: VM

Display the compiler context at compiling of the)context command. See the section Context for more details.

[])debug []

Routing: VM

This method activates debug mode in which greater detail of compiler activity is displayed. For example:

```
>debug

>2 3 + 4 *
Tags=[:numeric] Code="vm.push(2); "
Tags=[:numeric] Code="vm.push(3); "
Tags=[:stub] Code="vm.swap_pop._016(vm); "
Tags=[:numeric] Code="vm.push(4); "
Tags=[:stub] Code="vm.swap_pop._018(vm); "

>: double dup + ;
Tags=[:immediate] Code="vm._085(vm); "
  begin_compile_mode
Tags=[:macro] Code="vm.push(vm.peek()); "
Tags=[:stub] Code="vm.swap_pop._016(vm); "
Tags=[:immediate] Code="vm.context[:_316].does.call(vm); "
double => lambda {|vm| vm.push(vm.peek()); vm.swap_pop._016(vm);  }
  end_compile_mode

>5 double .
Tags=[:numeric] Code="vm.push(5); "
Tags=[] Code="vm._317(vm); "
Tags=[] Code="vm.pop._094(vm); "
10
```

This command is canceled by the)nodebug command.

[])elapsed []

Routing: VM

Display how much time has elapsed since the current virtual machine was started.

```
>)elapsed
Elapsed time is 606.454005 seconds
```

[])entries []

Routing: VM

Display the currently defined entries of the symbol table. See Appendix A for an example.

[])globals []

Routing: VM

Display the currently defined global values and variables and the strings they map to.

```
>42 val$: $zz

>)globals
$zz (_304)
```

[])irb []

Routing: VM

Launch into an interactive Ruby debug session:

```
>)irb

Starting an IRB console for fOOrth.
Enter quit to return to fOOrth.

irb(main):001:0>
```

This command is canceled by the IRB quit command.

[])load" … " []

Routing: VM

Load the file named in the embedded string. This unlike load” this method provides user feedback of the loading process.

```
>)load"docs/snippets/ugly_if"
Loading file: docs/snippets/ugly_if.foorth
Completed in 0.01 seconds
```

[])map" … " []

Routing: VM

Display the mapping information (if any) for the embedded string. See the section Exploring the Mapping System above for more information.

[])nodebug []

Routing: VM

Disable the debug mode described above.

[])noshow []

Routing: VM

Disable the show mode described below.

[])quit []

Routing: VM

Exit the fOOrth language system:

```
>)quit

Quit command received. Exiting fOOrth.

C:\Sites\fOOrth>
```

[])restart []

Routing: VM

Reset the virtual machine start time to now.

[])show []

Routing: VM

When this command is active, the contents of the stack are displayed after each command is processed.

```
>)show

>1 2 3

[ 1 2 3 ]
>+

[ 1 5 ]
>*

[ 5 ]
```

This command is canceled by the)noshow command.

[])start []

Routing: VM

Display the start time of the virtual machine.

```
>)start
Start time is 2015-05-10 17:26:25 -0400
```

[])threads []

Routing: VM

Display the currently executing threads in the fOOrth system.

```
>)threads
#<Thread:0x1aec3b0> vm = <Main>
```

[])time []

Routing: VM

Display the current time.

```
>)time
It is now: 2015-05-10 at 05:35pm
```

[])unmap" … " []

Routing: VM

Display the reverse mapping information (if any) for the embedded string. See the section Exploring the Mapping System above for more information.

[])version []

Routing: VM

Display the version string of the fOOrth language system.

```
>)version
fOOrth language system version = 0.0.6
```

[])vm []

Routing: VM

Display a virtual machine dump at execution of the)vm command. See the section Context for more details.

[])vm! []

Routing: VM

Display a virtual machine dump at compilation of the)vm command. See the section Context for more details.

[])words []

Routing: VM

Display the active methods defined for this virtual machine:

```
>)words
VirtualMachine Shared Methods =
!:          )nodebug    -infinity  ?dup       drop       over       true
"           )noshow     .:         [          dup        pause      try
)"          )quit       .::        accept     e          pi         tuck
)classes    )restart    .dump      accept"    epsilon    pick       val#:
)context    )show       .elapsed   begin      false      rational   val$:
)context!   )start      .restart   class:     if         rational!  var#:
)debug      )threads    .start     clear      infinity   rot        var$:
)elapsed    )time       .subclass: clone      load"      self       vm
)entries    )unmap"     .to_s      complex    max_float  space      {
)globals    )version    .vm_name   copy       min_float  spaces     {{
)irb        )vm         2drop      cr         nan        swap
)load"      )vm!        2dup       do         nil        switch
)map"       )words      :          dpr        nip        throw"
```

Appendix A – Symbol Glossary

```
Symbol Map Entries =
!                 .[]!              .denominator      .month
!:                .[]@              .do{{             .months
"                 .abs              .dump             .name
&&                .acos             .e**              .new
)"                .acosh            .each{{           .new_size
)classes          .alive?           .elapsed          .new_value
)context          .angle            .emit             .new_values
)context!         .append           .empty?           .new{{
)debug            .append{{         .even?            .numerator
)elapsed          .as_days          .exit             .odd?
)entries          .as_hours         .floor            .offset
)globals          .as_local         .fraction         .open
)irb              .as_minutes       .gcd              .open{{
)load"            .as_months        .get_all          .p2c
)map"             .as_seconds       .getc             .parent_class
)methods          .as_utc           .gets             .peek
)nodebug          .as_years         .hour             .pend
)noshow           .as_zone          .hours            .polar
)quit             .asin             .hypot            .pop
)restart          .asinh            .imaginary        .posn
)show             .atan             .init             .pp
)start            .atan2            .intervals        .push
)stubs            .atanh            .is_class?        .r2d
)threads          .c2p              .join             .rationalize_to
)time             .call             .keys             .real
)unmap"           .call_v           .labels           .restart
)version          .call_vx          .largest_interval .reverse
)vm               .call_with        .lcm              .right
)vm!              .call_x           .left             .right?
)words            .cbrt             .left?            .rjust
*                 .ceil             .length           .round
**                .cjust            .lines            .round_to
+                 .class            .list             .rstrip
-                 .clear            .ljust            .sec_frac
-infinity         .clone            .ln               .second
.                 .clone_exclude    .load             .seconds
."                .close            .lock             .select{{
.+left            .conjugate        .log10            .shell
.+mid             .contains?        .log2             .shuffle
.+midlr           .copy             .lstrip           .sin
.+right           .cos              .magnitude        .sinh
.-left            .cosh             .main             .sleep
.-mid             .cr               .map{{            .sort
.-midlr           .create           .max              .space
.-right           .create{{         .mid              .spaces
.1/x              .cube             .mid?             .split
.10**             .current          .midlr            .sqr
.2**              .d2r              .min              .sqrt
.:                .day              .minute           .start
.::               .days             .minutes          .start_named
```

```
.status          1+               VirtualMachine   nil=
.strip           1-               [                nip
.strlen          2*               ^^               not
.strmax          2+               a_day            now
.strmax2         2-               a_minute         or
.subclass:       2/               a_month          over
.tan             2drop            a_second         p!"
.tanh            2dup             a_year           p"
.throw           :                accept           parse
.time_s          <                accept"          parse!
.to_a            <<               an_hour          pause
.to_duration     <=               and              pi
.to_duration!    <=>              begin            pick
.to_f            <>               class:           rational
.to_f!           =                clear            rational!
.to_i            >                clone            rot
.to_i!           >=               com              self
.to_lower        >>               complex          space
.to_n            ?dup             copy             spaces
.to_n!           @                cr               swap
.to_r            Array            distinct?        switch
.to_r!           Bignum           do               throw"
.to_s            Class            dpr              true
.to_t            Complex          drop             try
.to_t!           Duration         dup              tuck
.to_upper        FalseClass       e                val#:
.to_x            Fixnum           epsilon          val$:
.to_x!           Float            f"               var#:
.unlock          Hash             false            var$:
.utc?            InStream         format           vm
.values          Integer          identical?       xor
.vm              Mutex            if               {
.vm_name         NilClass         infinity         {{
.with{{          Numeric          load"            ||
.year            Object           local_offset     ~
.years           OutStream        max              ~"
/                Procedure        max_float        ~cr
0<               Queue            min              ~emit
0<=              Rational         min_float        ~getc
0<=>             Stack            mod              ~gets
0<>              String           nan              ~space
0=               Thread           neg              ~spaces
0>               Time             nil
0>=              TrueClass        nil<>
```

Appendix B – Regular Expressions

Elements of syntax in this guide are expressed using regular expressions. While regular expressions are powerful and concise, they are also cryptic and non-intuitive. This summary provides a brief overview of those subset of regular expression elements used in this guide. It only provides a thread-bare overview of the subject of regular expressions and the reader is encouraged to seek further information on this complex subject.[100]

Expression Character	Description
`.`	Any character except newline
`[A-Z]`	Any uppercase alphabetical character.
`[A-Za-z0-9_]`	Any alphanumeric or underscore character.
`\.`	A single '.' character.
`\/`	A single '/' character
`\$`	A single '$' character.
`\d`	A digit (0..9)
`X*`	0 or more repetitions of the X expression.
`X+`	1 or more repetitions of the X expression.
`^`	The beginning of the line or string.
`$`	The end of the line or string
other	Non-regex commands are expressions for themselves.

100For an excellent, interactive, free regular expression development tool see Rubular at http://rubular.com/

Appendix C – Git

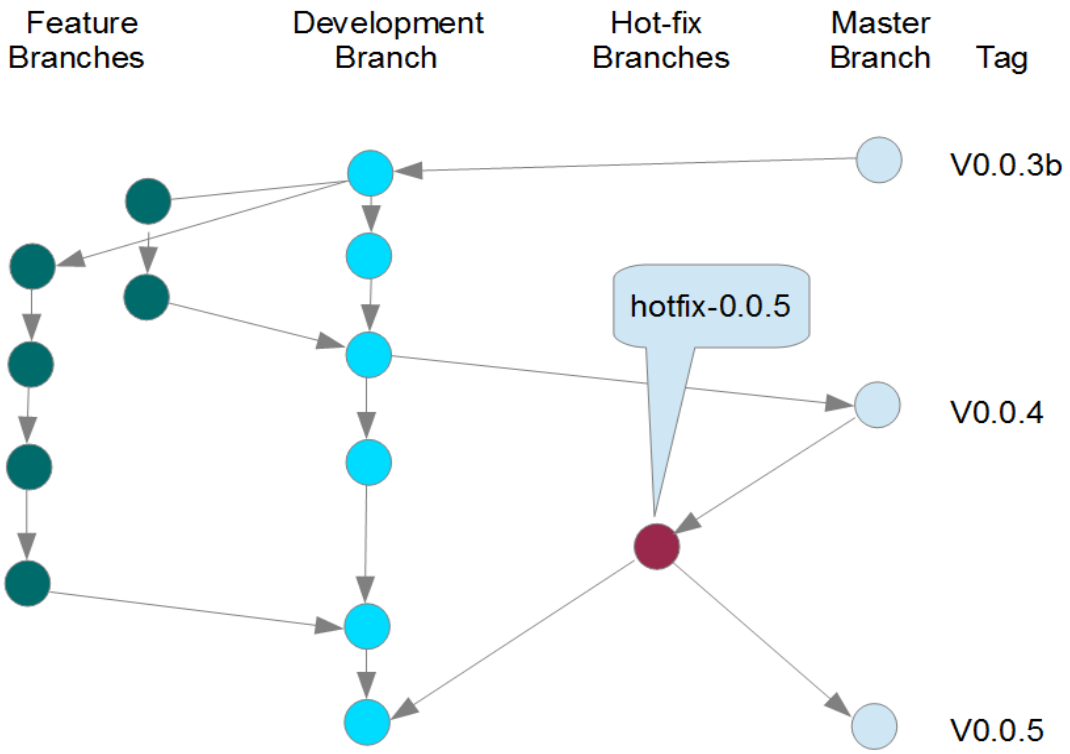
The fOOrth project's source code is managed through the git source version management system. Prior to version 0.0.4, all work, bug-fixes, updates, and improvements where all done on the master branch. Moving forward, this is not a suitable model for further development and a better, more manageable system is required.

The system adopted by the fOOrth project is that laid out in the blog “A successful Git branching model[101]”. Under this model, the master branch is reserved for ready-to-deliver code. All development efforts are shifted to the development branch.

Efforts of a short duration, like simple bug fixes, can be done directly on the development branch. Those with a longer duration or more uncertain outcome should be branched from the development branch with an appropriate descriptive name. These development branches are merged[102] back into the development branch or dropped as appropriate.

The development branch will be merged into the master branch to release new code. A release branch may be employed if the release complexity warrants using one.

Hot-fixes branch from the master and merges to both the master and development branches.



101This may be found at: http://nvie.com/posts/a-successful-git-branching-model/
102Git merges should always be done with the '--no-ff' option to avoid losing branch history information.

Index

Edit History:

PCC – December 14, 2014 – Initial draft of User's Guide started.

PCC – May 14, 2015 – Completed the initial draft.

PCC – May 26, 2015 – Added revisions developed during sabbatical.

PCC – May 28, 2015 – Added Appendix C – Git

PCC – June 2, 2015 – Added more detailed descriptions of Array and Hash literals.

PCC – June 11, 2015 – V0.1.0 Major changes for procedure literal support.

PCC – June 15, 2015 – V0.2.0 Adding Mutex and Time classes.

PCC – July 25, 2015 – V0.3.0 Adding the Duration class.

– Stuff stored here for use throughout. Delete someday.

[before] word [after] Routing: description	
Code	**Result**

Local Methods:

[before] word [after] (h4) Routing: Compiler Context. description	
Code	**Result**