

Introduction



Welcome to Final!

Final is a new, modern, stack-based programming language designed by [Brad Aisa](#) in the tradition of Forth. “But this ain’t yer granddaddy’s Forth!” Final has all the features a modern programming language needs, such as:

- rich primitive types
- polymorphic operators
- Objects and Types
- Compound classes such as Lists, Maps, Sets, and Dynamic Objects
- Vectors, Matrices, and statistical and model-fitting/predicting operations
- Structured Exception Handling (SEH)
- Dynamically loadable modules (work in progress, local modules only this release)
- support for Http/Web, Json, Threading
- Blitting and Drawing onto a Pixmap for creating images such as PNG/JPEG/GIF
- a final-gui version with support for creating Graphical User Interfaces
- a Simulation/Game framework with animation support, and a Motion/Hit engine
- ... and much more

Final was written primarily to be a language for the creation of AI Tools, but it is just as useful as a general purpose stack-based programming language. Final can be run in an interactive Terminal User Interface (TUI), a graphical interface (GUI), or can be embedded in a Request/Response mode in other Go programs.

Final runs on Windows, Mac, and Linux, for both AMD64 and ARM64 cpus.

A custom version of Final for Raspberry Pi 5 (64-bit Pi OS) also exists, with a library of low-level hardware words. (Basic pin set and read in this version, full peripheral support in the next release.)

You’ll get the most out of this introductory tutorial and user guide by installing Final and following along. So let’s get started...

Installation

Final is available for Windows, Mac, and Linux. This version, v0.0.8 provides support for:

- macOS Apple Silicon (arm64)
- macOS Intel (amd64)
- Windows x64 (amd64), and
- Linux x64 (amd64).

Raspberry Pi and other CPU variants will be included in an upcoming release.

Final is easy to install, and easy to remove, should you wish to do so. It is provided as a simple .zip file, for each supported platform/cpu combination.

Install instructions

1. Go to the downloads page, and download the .zip for your platform/cpu.
2. Move the .zip to your home folder.
3. Double-click the .zip in your UI, or cd to your home folder and run the unzip program for your OS.
4. This should create a folder `Final` with the `final` and `final-gui` binaries (`final.exe` / `final-gui.exe` on Windows) in the root, and Final's bootstrap files, and example programs.
5. Launch that file in your UI.
6. (optional) Select one of the examples in `Final/examples` and open it in your favorite editor; associate the .fnl extension with your editor.

Troubleshooting

Prefer launching from inside the unzipped `Final/` folder. From a shell, `cd` into `Final/` then run `./final` or `./final-gui` (`final.exe` or `final-gui.exe` on Windows).

Uninstall instructions

1. Delete the folder `Final` in your home folder. That's it!

The Final TUI

When you launch `final` from the command line or your GUI you should see the following Terminal User Interface TUI (with empty text boxes.) You can resize your terminal, and the TUI will automatically resize. (Note: the TUI will also work, with some limitations, over ssh.)



Let's take a quick look at the TUI, starting from the bottom:

Shortcut Bar This bar shows the shortcut key combinations for the most used commands. Mac users: note that Ctrl is used instead of the Cmd key.

Prompt Area This is where you enter Final commands, or code. `Enter` sends the current text in the area. You can use Ctrl-J to add new lines without sending. And if you Paste text from an editor, with newlines, it won't send until you hit `Enter` or press `Send`.

History Area This shows a running list of the prompts you sent. You can copy code from this text area.

Response Area This shows the result of a Send. If the result contained no text, "(ok)" is printed, as confirmation. Just like History, you can copy text from this text area.

Menu Bar This contains 2 particularly useful commands:

File/Load – let's you browse for a .fml file, which is loaded (compiled); Help/Words – this will get you started with Final's built-in help

Example conventions used in this guide

Code examples in this guide do not try to depict the TUI—this would be both wasteful of screen space, and impossible to copy from. Instead, we use a pseudo bare command line interface that looks like this:

```
> 1 2 + $
```

```
3
```

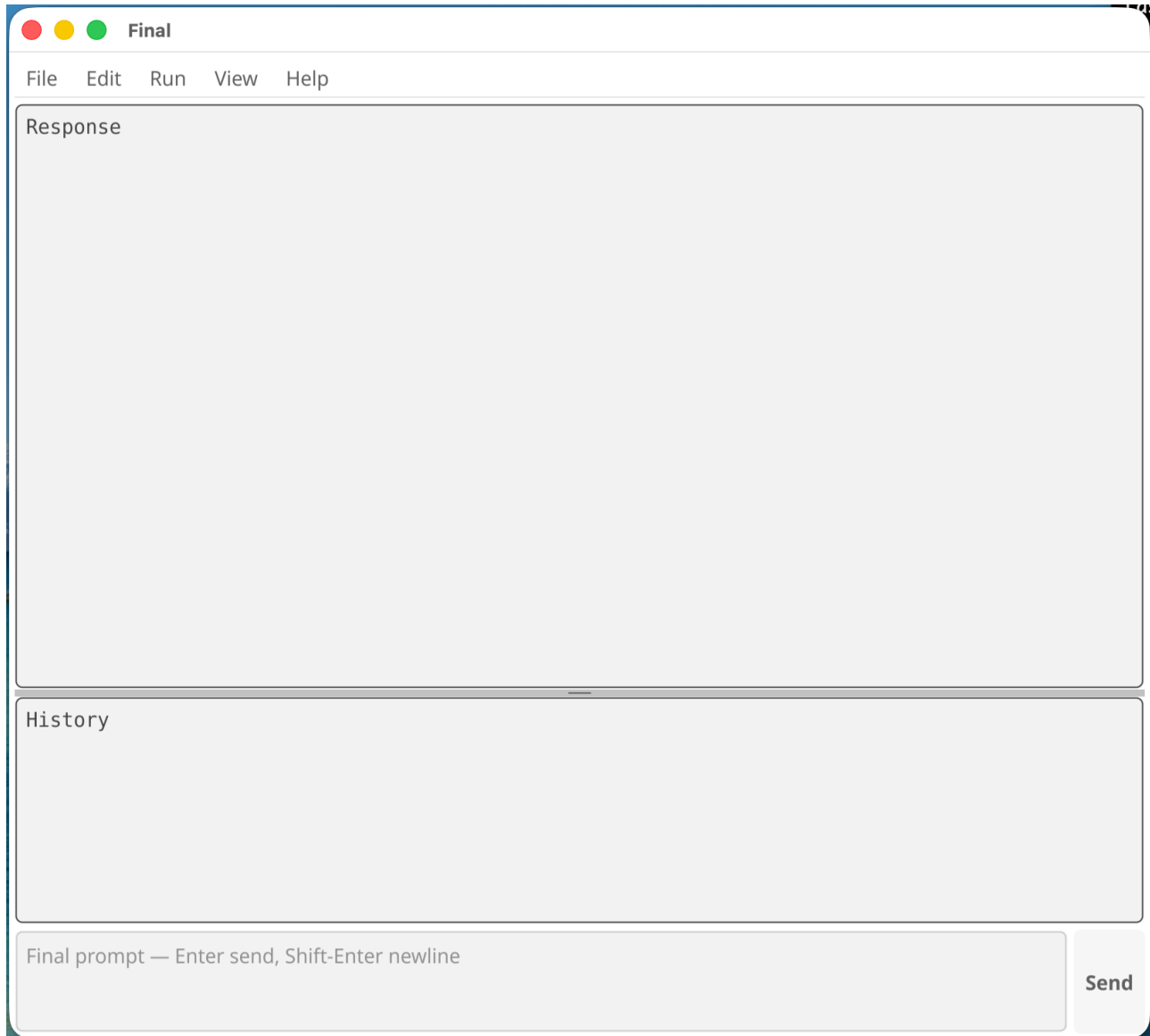
- the text after > and any non-empty lines below are the prompt
- a blank line
- the expected output of the prompt, which may be many lines; or (ok) if no output is expected

Working along with the examples, you can either type into the TUI Prompt area, or copy/paste.

On to some code!

The Final GUI version

When you launch `final-gui` from the command line or your GUI you should see the following Graphical User Interface TUI (with empty text boxes.) You can resize the Window. (Note: for ssh, use the `final` binary; it will work, with some limitations, over ssh.)



Let's take a quick look at the GUI, starting from the bottom:

Prompt Area This is where you enter Final commands, or code. `Enter` sends the current text in the area. You can use Shift-Enter to add new lines without sending. And if you Paste text from an editor, with newlines, it won't send until you hit `Enter` or press `Send`.

History Area This shows a running list of the prompts you sent. You can copy code from this text area.

Response Area This shows the result of a Send. If the result contained no text, “(ok)” is printed, as confirmation. Just like History, you can copy text from this text area.

Menu Bar This contains three particularly useful commands:

File/Load – let’s you browse for a .fml file, which is loaded (compiled); View – Prompt/History (Response and History) or Stdout/Stderr (standing `$ / $$`); the current view’s menu item is disabled; Help/Words – this will get you started with Final’s built-in help.

Example conventions used in this guide

Code examples in this guide do not try to depict the GUI–this would be both wasteful of screen space, and impossible to copy from. Instead, we use a pseudo bare command line interface that looks like this:

```
> 1 2 + $
```

```
3
```

- the text after > and any non-empty lines below are the prompt
- a blank line
- the expected output of the prompt, which may be many lines; or (ok) if no output is expected

Working along with the examples, you can either type into the GUI Prompt area, or copy/paste.

On to some code!

Hello Final!

When you first launch `final` you are in an interactive interpreter. You can enter numbers and strings, operators, or words, and Final will execute it according to its interpretation rules (which are covered later in [Interpreter](#)).

So one of the first things typically done when introducing a new language is the venerable “Hello World!” example. Let’s do it interactively first. Try the following:

```
> "Hello from Final!\n" $
Hello from Final!
```

Your string literal got pushed on DS (see below), then `$` “print” printed it out. Easy, peasy!

Let’s try a simple arithmetic example. Enter the following, and observe the result:

```
> 1 2 + $
3
```

A central construct in Final is the Data Stack (DS). It has no counterpart in conventional languages. Conventional languages use named parameters in functions to pass parameters, and sometimes global variables.

Final uses the DS to hold intermediate operands, pass operands to Final’s counterpart of a function, called a `word`, and return results. When you write a literal value in the interpreter, it gets pushed onto DS.

Some may notice the arithmetic rules in Final work the same as in HP calculators. It is called Reverse Polish Notation (RPN). A nice feature of RPN is that you do not need grouping operators like `( )` to group and order operations, any such expression can be expressed using the RPN/stack-based approach of Final.

This way of thinking does take a bit of getting used to, but it is the “secret sauce” that makes all Final-like languages so uniquely flexible and efficient, in terms of code factoring, and code reusability.

Try out a more complicated example. This requires parentheses in a conventional language, due to order of operations and operator binding. You do it in Final by working inside-out, then left to right at each grouping level: $((3 + 1) * (11 - 6)) / 4$. Tip: you enter operands for things like - and / in reading order, so: $a - b$ is entered as “a b -”.

```
> 3 1 + 11 6 - * 8 / $
2
```

If you have a sharp eye, you will know that the result is actually 2.5, not 2. The cause is Final’s type coercion rules. If all the input operands to arithmetic operators are integers, the result will be an integer, due to integer truncation of rounding. To make it be a float, just change one of the operands to include a decimal point. This causes it to be interpreted as a float, and Final’s coercion rules always promote the result of mixed int/float operations to have a float result, so the “float virality” will propagate through. Try it!

```
> 3 1.0 + 11 6 - * 8 / $
2.5
```

Now let’s write our first word in Final. Words are stored in something called a **Dictionary**; there are at least 2 dictionaries when you start Final:

- the sealed **core** Dictionary that contains the words of the Final language
- a (mostly) empty **local** dictionary in your Session, into which you can add new definitions

Let’s write one now, enter this:

```
> : helloFromFinal "Hello from Final!\n" $ ;
helloFromFinal

Hello from Final!
```

You just created your first Final word called `helloFromFinal`, and then invoked it! Here is the anatomy of what is called a **colon definition**

1. opening **:** then a space; this puts Final into compiling mode
2. the name of the new word

3. the literals, operators, and words making up the definition
4. `;` closes the definition and adds it to your local dictionary

The `helloFromFinal` following the definition then invokes it, causing the word to print the text.

Creating new words is as simple as that. When you create new words, you can include words you have previously defined along with core words.

Simple Input

Final doesn't provide a way to directly prompt for input from the Prompt area, but it has a simple API for raising a modal dialog to either ask for a confirmation, or request a few fields of information. We provide a quick example of the latter here, and cover the topic in more detail in the [Input/UI Dialogs](#) topic.

No introduction would be complete with some Monty Python. The following example is in `examples/dialog/ask-demo.fnl`.

```
: ask-demo
( "What is your name?" "What is your quest?" )
Dialog.ask^ if
  "Nice to meet you, %s. Off you go - \"%s\" is a fine quest.\n" fmt $
else
  "Then we shan't proceed. Nobody expects the Spanish Inquisition.\n" $
then
;
```

The code provides the requested field labels using the convenient Paren operators for building a List. It then asks Dialog to post the dialog via the `ask` method, which returns one of two types of result shape:

- `true` then a List of the replies, in order of request
- `false` and no List

You test for the response type using an `if` statement, and if `true` can then pass the List directly to a `fmt` statement and print it out. If `false`, you can print a default reply, as above.

`Dialog` is covered in more detail in [Input/UI Dialogs](#), Paren operators in [Compound Builders](#) and `fmt` in [Output Formatting](#).

Let's now move on, and dig a little deeper into Final...

Basic Types

Final has a set of basic, primitive types, and a rich set of collection and other compound object types. We'll cover the basic types here.

Basic types are immutable, meaning that once created, their value does not change. In operations using the value and returning a value, a new value is created; you cannot alter or change an instance of a basic type.

Null

The Null type has only one value: `null`. It exists primarily to support Json. It's use otherwise is discouraged. Final has a pattern it widely uses for indicating there is "no value" returned from an operation, so Null is not used by Final, except when generating or parsing Json, or initializing strongly typed fields with no value yet.

Bool

The Bool type has the values `true` and `false`. It is not a numeric type. However, Final has the concept of "Truthiness", which enables some of the other types to be used as if they were Bool.

Numeric types

Final has two integral types, and one real type.

- `Int` - a 64 bit signed integral value;
- `UInt` - a 64 bit unsigned integral value;
- `Float` - a 64 bit IEEE-754 floating point value;
- `Rune` - a synonym for `Int`, representing a 32 bit unicode character.

`Int` is the type you should use for most integral operations and representations, even in contexts such as an ordinal index, in which negatives do not occur. Almost all of Final's apis that use or return an integral value use `Int`.

`UInt` is provided primarily for contexts which require integral values larger than $2^{63}-1$, or for flags or bitfields that use bit 63, in which you don't want the natural printing to default to displaying a negative, 2's complement interpretation. Note that `Int` and `UInt` do not interoperate in dyadic operations; type coercion is covered in the next topic. You create a `UInt` literal by appending an integral value with the `u` suffix.

`Rune` is a synonym for `Int`, and is used in contexts involving Unicode code points, called `runes`. A Unicode rune is a 32 bit integer value.

String

A Final `String` is an immutable sequence of Unicode characters using Utf-8 encoding. All mutating operations on strings accept immutable string inputs, and return a new immutable string value. (Final provides mutable `StringBuilder` to successively build a `String` from parts.) Strings are not 0-terminated. Strings are copied by reference throughout the runtime, so there is only one underlying value, regardless how many times it is moved or copied.

Utf-8 encoding provides an excellent compromise between byte size, efficiency, and interoperability with the Web and Json, as well as other benefits. ASCII-7, i.e. 7-bit ASCII is a proper subset of Utf-8, so an ASCII string in bytes is identical in Utf-8. Unicode runes beyond ASCII are encoded in either 2, 3, or 4 byte sequences, with the majority of Western/Latin characters only needing 2 or 3 bytes. It is thus more storage efficient than Utf-32, which uses a fixed 4 bytes per character.

One caution: the concept of “length” in Unicode in general, and especially Utf-8 requires decoding the entire string, so Final does not provide a true string length. The `String.len` property is the length in bytes of the encoding. If the string is known to contain only ASCII-7, then that is also the length. Final provides an efficient mechanism to iterate the Runes of a String, which is explained in the [String](#) topic.

Any

`Any` is a type that can hold any other Final type, basic or compound/object. The DS is composed of a growable stack of `Any` values. So when a concrete type goes on DS, it will always be wrapped in an `Any`.

“Truthiness”

For convenience, you can use a value of the following types in most operations expecting a `Bool`, such as boolean logic operators, and `if` and `case` control structures.

- Numeric types: exact 0 is false, non-0 is true
- String: empty `""` is false, anything else is true

No other types are accepted, and will raise a runtime error if used.

DateTime and Interval

These are also basic types, but discussion is deferred to [DateTime and Interval](#).

Let's move on to a discussion and demonstrations of the Final Interpreter, which includes details on expressing literal values of the basic types in the interpreter and compiled words.

Interpreter

When you first start the Final TUI, Final is in interpreter mode. This is an interactive request/response type of interface. When you start a colon definition with `:` Final goes into compiling mode. We will discuss the basic lexical rules of Final text, which applies both to the interpreter and compiler, then cover the interpreter itself.

Final Lexical Rules

Final source is Utf-8, of which regular ASCII-7 is a proper subset. Note that Final does not accept “Code Pages”, which are the early DOS and Windows era “hack” for creating overlays in the top 128 bits of ASCII. Such sources will be mis-interpreted by Final.

Final text is parsed as tokens, which are whitespace separated values as defined by the parsing rules. Individual Final operators and words must be separated by whitespace, unlike languages which permit whitespace eliding in many contexts. (The one exception to this is Final’s compound object path syntax, covered later.)

Valid whitespace is: space, tab (“\t”), newline (“\n” aka linefeed), or carriage return (“\r” aka return.) In an operation counting or splitting lines, Final accepts “\r\n” Windows-style newlines to be a single end-of-line marker. It is suggested that you uniformly use the single “\n” form for both Final source files, and textual data files. No other whitespace, including Unicode whitespace characters, are accepted.

These are the types of tokens in Final

- Null literal: `null` ;
- Bool literal: `true` or `false` in lowercase;
- Int literal: any valid decimal, octal, hex, or binary number, without a . decimal
eg, “10” “0x7F” “0b101010” “-49”
note: numbers may not be preceded with `+`, only `-` ;
- Uint literal: a valid positive integer, with “u” affix, eg. 25u;
- Float literal: a valid number, that contains a decimal point; exponential notation is allowed;
- Quoted String: a textual value enclosed in quotes; Go-style escapes `\n` `\t` `\xHH` `\uHHHH` `\UHHHHHHHH` ; literal Unicode (emoji, non-English) is accepted in the source unless it is a dangerous character (see below). Dangerous characters may appear in the string *value* only via `\x` / `\u` / `\U` ;
- Raw String: a textual value enclosed in back ticks: `the rain in spain`; the entire text is interpreted literally, including line breaks; no escapes. Dangerous Unicode

in a raw string is a parse error;

- Unicode Rune: a single rune in single quotes; printable characters, C-style escapes, `\xHH`, `\uHHHH`, `\UHHHHHHHH`;
- `//` Comment: the `//` characters signify a comment, to the end of the line;
- `/* */` Comment: a comment that can span multiple lines; it can include embedded `//` comments (which are just considered part of the whole comment text) but cannot include nested slash-star comments;
- Compound Path accessor: a valid compound path accessor (covered later);
- an identifier, which is either the name of an existing word, or based on the previous operator such as `var:` will become a new word; identifiers may not start with a number, and must be printable Unicode characters.

Identifiers

Identifiers in Final are far more permissive than in most other languages—they can contain combinations of letters, numbers, and most special characters.

Most of the special characters are used by Final to convey a family of related operations, so before going wild and using them in your identifiers, take the time to learn the lexicon, and ask if your new word belongs in that group.

Some characters and patterns are reserved by Final, and may not be used in identifiers. Here is a summary of the rules, or disallowed characters:

- an identifier cannot begin with a number
- the characters must be Printable/non-whitespace Unicode characters
- the following characters cannot be used: `.` `@` `!` `\` `^` `~` `"` ``` `'`
- Source text may not contain “Dangerous” characters: Trojan-Source / invisible Unicode (Bidi marks, zero-width characters, variation selectors, and similar). The parser stops at that code prints out the value in hex. If you absolutely must use those characters, put them in a quoted string or rune using `\x`, `\u`, or `\U`.

Naming Guidelines

Final follows a standard of using camelCase for most identifiers. Exceptions are Types and constants, which are in PascalCase. You may also choose to use hyphen-naming instead of camelCase if you wish.

Final encourages the use of code factoring to keep individual words small, and to convey the purpose of a piece of code by way of its name, as opposed to relying on comments. To avoid name collisions in your User dictionary, it is suggested to factor

by keeping the main word name, then an underscore, followed by the factored name. **Subwords** within a word, are also supported, and often more convenient.

The Final Interpreter

Final's interpreter uses simple rules. It skips whitespace, then reads a token. Based on the type of the token, it does the following:

1. Comment: discard it
2. Literal: push it on DS
3. Identifier: look that up in the current Dictionary search path if it is found, then the word is executed
4. else signal an error

If the word is `:` then Final is put into Compiling state. It leaves compiling state and re-enters interpreting state after the `;`.

So for you, the user, this is very convenient, because you can use the interpreter for interactive purposes, including compiling and testing new words. Likewise, a Final source file can intermix immediate, interpretive behavior with compilation of new words. This has a number of useful use-cases, which we will mention as we get to the features involved.

Let's take a quick look at compiling state, and how Final compiles new words. Then we'll move on to a quick intro to Final's control flow blocks, and its collection types.

Compiler

The Final compiler creates new compiled words from existing words. Final code gets compiled into a Code Block (CT) table that contains the OpCodes of Final axioms, as well as some prefix modifiers, and immediate values to an OpCode, branches, and so on. The code is then executed in a tight run loop with an IP pointing the current instruction. OpCodes get dispatched via an efficient Go-compiled jump table. The resulting execution is highly efficient and fast, and for some operations can rival the performance of natively compiled machine code, due to CPU code and data caching. For details on how Final works, see [How Final Works](#).

Final Compiler

The Final Compiler is used to compile a new colon word. It follows rules somewhat parallel to that of the Interpreter. Here is how it handles tokens;

- Comments are dropped and ignored.
- A literal value is compiled into the code as a literal; at runtime, the value gets pushed on the DS
- a compound path accessor is broken into its parts, and compiled into the CT
- an identifier is looked up as a word
 - if found, it is compiled into the word as a Nest call to that word's DC/CT index
 - if not found, an error is raised, and compilation stops

Some words, such as control flow words, have a more complex compile-time behavior; these words combine immediate interpretive behavior, along with emission of compiled code. For example, control flow constructs need to keep track of the IP address during compilation, place dummy values into the CT that will hold branch addresses, then resolve those branches when closing words are reached. But the outcome is the same: directly executed compiled code is what executes at runtime.

Speaking of control flow words.. enough theory! Let's move on to a quick overview of Basic Operators, then from there to control flow words, along with examples.

Basic Operators

Final has the basic set of operators that most languages provide, for things like arithmetic, boolean logic, bitwise operations, compares, and so on. Each family is briefly described below, with the operators. Type `help <op>` in the prompt to see a description of the operator, and its stack effects.

The type coercion/promotion rules are also listed for each family. A decision has been made in Final to not try to coerce/promote arithmetic operations mixing `Int` and `Uint`. The reason is that there are many cases where coercion cannot be inferred. The user must decide how the two values are to be interpreted, and apply the `>Int` or `>Uint` bitwise coercion operator that makes sense in the context. Mixing the two types in many operators causes a runtime error. If you are unsure, try mixed values first in the interpreter, before including it in your code.

Try out some of the operators in each category in the UI!

Arithmetic

Dyadic

`+` `-` `*` `/` `%` `modrem`

Monadic

`abs`

Promotion / coercion

- Same type → result is that type (`Int / Int` , `Uint / Uint` , `Float / Float`).
- Any operand is `Float` → the other is converted to `Float` ; result is `Float` .
- Mixed `Int` and `Uint` → **runtime error**. Cast one side with `>Int` or `>Uint` first.
- `%` and `modrem` require matching integral types (`Int / Int` or `Uint / Uint`); mixed `Int / Uint` → **runtime error**.

Boolean

Dyadic

`&&` `||`

Monadic

`not`

Promotion / coercion

- Operands must be **Truthy**: `Bool`, `Int`, `Uint`, `Float`, or `String`.
- Numeric zero and empty `String` are false; anything else in that set is true; `Bool` is itself.
- Other types (lists, maps, objects, ...) → **runtime error**.

Bitwise

Dyadic

`&` `|` `xor` `bitset` `bitclear` `bittoggle`

`<<` – logical shift left

`>>` – arithmetic shift right

`rotl` `rotr` – rotate left/right

Monadic

`bitnot`

Promotion / coercion

- Operands must be integral (`Int` or `Uint`); non-integral → **runtime error**.
- Dyadic `&` `|` `xor`: both the same type → that type; otherwise result is `Int` (any `Uint` is reinterpreted as `Int`). Never auto-promotes `Int` → `Uint`.
- `bitnot`: result matches the operand's type.
- Flag mutators (`bitset` `bitclear` `bittoggle`): result matches the flags value being updated.
- Shifts and rotates: result matches the value being shifted; the count may be `Int` or `Uint` (non-negative). Counts are automatically capped by the runtime to 64 bits.

Equality and Comparison

Dyadic

`==` `<>` `<` `<=` `>` `>=`

Monadic

`==0` `<>0` `<0` `<=0` `>0` `>=0`

Promotion / coercion

- Result is always `Bool`.
- Both operands must be in the **same family**, or → **runtime error**:
 - **String** — both `String` (ordering is Unicode byte ordering, see: [Strings](#)).
 - **Numeric** — both numeric (`Int` / `Uint` / `Float`), with the same promotion rules as arithmetic (including: mixed `Int` / `Uint` → **runtime error**).
 - **Truthy** — both Truthy, but not both Numeric and not both String (each coerced to `Bool`, then compared).
- Examples: `1 true ==` is fine (Truthy); `1.0 1u ==` is Numeric; `"a" 1 ==` → **runtime error**.
- Zero-compare forms push a typed `0` matching the value (`Int` / `Uint` / `Float`), then compare — so `uint` stays unsigned.

Casts

Use these when you need an explicit type change or a membership test. The `?` forms leave a success flag instead of raising an error; the forms without `?` raise an error on failure. Details: [help](#) on each word.

Type probes

`Null?` `Bool?` `Int?` `Uint?` `Float?` `String?`

Exact runtime type only — no conversion. Leave a `Bool`. (value – is:b)

Representation casts

`>Bool` `>Int` `>Uint` `>Float`
`>Bool?` `>Int?` `>Uint?` `>Float?`

Compatible primitives only — **no** string parsing.

Semantic casts

`>>Null` `>>Bool` `>>Int` `>>Uint` `>>Float` `>>String`
`>>Null?` `>>Bool?` `>>Int?` `>>Uint?` `>>Float?` `>>String?`

Like representation casts, plus string parsing where it applies (e.g. `"42" >>Int`).

Trancendentals and Related

Final provides a set of transcendental operators and constants, such as `sin`, `cos`, `Math.pi` etc. The operators take any type of numeric value, and typically return a `Float`. Enter `list-math` for the list, then `help` on each operator for its description and stack effects.

`Float` provides some useful constants like `Float.NaN` and tests like `Float.isNaN^` and `Float.isInf^` to help deal with the thorny corner cases of IEEE-754 NaNs and Infs. use `help Float` to list them.

Monadic

Op	Description
<code>acos</code>	arccosine; result in radians
<code>acosh</code>	inverse hyperbolic cosine
<code>asin</code>	arcsine; result in radians
<code>asinh</code>	inverse hyperbolic sine
<code>atan</code>	arctangent; result in radians
<code>atanh</code>	inverse hyperbolic tangent
<code>cbrt</code>	cube root
<code>ceil</code>	smallest integer <code>Float</code> not less than the argument
<code>cos</code>	cosine of radians argument
<code>cosh</code>	hyperbolic cosine
<code>exp</code>	e raised to the power of the argument
<code>floor</code>	largest integer <code>Float</code> not greater than the argument
<code>log</code>	natural logarithm (base e)
<code>log10</code>	base-10 logarithm
<code>log2</code>	base-2 logarithm
<code>round</code>	nearest integer <code>Float</code> (half away from zero)
<code>sin</code>	sine of radians argument
<code>sinh</code>	hyperbolic sine
<code>sqrt</code>	square root

Op	Description
<code>tan</code>	tangent of radians argument
<code>tanh</code>	hyperbolic tangent
<code>trunc</code>	truncate toward zero to an integer Float

Dyadic

Op	Description
<code>pow</code>	raise x to the power y (<code>x:n y:n -- f</code>)

Constants and Helpers on Float

Const/Op	Stack	Description
<code>.maxFloat</code>	(<code>- f</code>)	Largest finite Float
<code>.minNonZero</code>	(<code>- f</code>)	Smallest positive non-zero
<code>.+Inf</code>	(<code>- f</code>)	IEEE-754 positive Infinity sentinel (1)
<code>.-Inf</code>	(<code>- f</code>)	IEEE-754 negative Infinity sentinel (1)
<code>.NaN</code>	(<code>- NaN:f</code>)	IEEE-754 Not A Number sentinel (2)
<code>.isInf</code>	(<code>a:n - b</code>)	true if the value is an IEEE-754 <code>+Inf</code> or <code>-Inf</code> value
<code>.isNaN</code>	(<code>a:n - b</code>)	true if the value is an IEEE-754 NaN

Notes:

1. Can be compared to valid Float values; use `.isInf` to test for `+Inf`;
2. `.NaN` is only for producing the NaN sentinel, if needed; use `.isNaN` to test for NaN; all float operations using NaN as an input return NaN; `NaN <> NaN` in comparisons

Constants on Math

Const	Stack	Description
<code>.e</code>	(<code>- f</code>)	Base of natural logarithms (e)
<code>.pi</code>	(<code>- f</code>)	Pi
<code>.ln2</code>	(<code>- f</code>)	Natural log of 2
<code>.ln10</code>	(<code>- f</code>)	Natural log of 10

Control Flow

A set of Control Flow words are used to create control flow blocks. Final has a rich set of control flow blocks. All Final Control Flow blocks must be used in a colon definition; you will get an error if you try to use any of these in the interpreter.

Before diving in to the blocks, this is a good time to introduce Final's convention for describing the stack effects of a word, called its "context effects". The convention follows simple rules for writing a comment that describe the state of the stack on input, and its shape on output. When placing a comment before a word, it is a good idea to use the `/* */` form, and place the context effect line last. The compiler picks up a last comment before a word, and stores it in the `.doc` field of the Word. This lets you print it by going `help wordname`. This is the comment for `+`

```
/*
Add two numbers, and return the sum.
( a:n b:n -- c:n )
*/
: + ... etc.
```

The `( )` indicate a stack effect. The left hand side lists the expected input values, if any. you can optionally give them a mnemonic name, eg `a` and `b` here, a colon, and the type of the value. There are codes for common values:

- `b` or `t` Bool or Truthy, most Bool contexts take Truthy (note: Json requires Bool only)
- `i` Int
- `u` UInt
- `iu` integral, Int or UInt
- `f` Float
- `n` numeric, any of Int UInt Float
- `s` String
- `a` Any

For other types, such as objects, collections, etc. the convention is to write out the entire typename. There are a few shortcuts for common types, such as `DO` for `DynamicObject`. This is covered in more detail in [Word Doc Guidelines](#)

Let's dive in to the blocks. We introduce each with an example that you can enter and try.

- Definite:
 - `if else then`
 - `switch ... /switch`
- Loops:
 - infinite: `forever ... /forever`
 - iterated: `foreach ... /foreach`
 - pretest: `begin ... while ... repeat`
 - post-test: `begin ... until` (also: `begin ... again` for a legacy infinite loop)
- Break and Continue
 - `break: break`
 - `continue: continue`

Definite Blocks

`if / else / then`

Takes a Truthy value. The `else` arm is optional — skip it when you only care about the true path.

```
> // ( yes:b -- )
: announce
  if "yes!\n" $ else "nope.\n" $ then
;
true announce

yes!
> false announce

nope.
```

`switch`

The `switch` statement is Final's pattern matching statement. It takes a value, called the target, and any number of `case` clauses that expect a truthy value. You may use any kind of comparison or matching operation for each case. The target value provided at the start is available anywhere within the `switch` using the keyword `a`. If you nest switches, you can access the target value of up to two outer containing switches with `b` and `c`.

Typically, each case expression will start with `a` followed by an equality or similar matching statement. However you are not required to use the target in any arm, enabling you to mix pattern matching with other logic. For strings, in addition to

simple `==` statements, you could also use Regex, or any of the `str::` expressions, such as `str::indexOf`.

The case clauses are executed in lexical order, until a case matches; when that `case` clause reaches `/case` the `switch` ends, and control is passed to the next statement after `/switch`. An optional `default` clause will execute if no case matches.

```
> // ( choice:i -- )
: pick
  switch
    a 1 == case "got 1\n" $ /case
    a 2 == case "got 2\n" $ /case
    default
      "got something else\n" $
    /default
  /switch
;
1 pick

got 1
> 99 pick

got something else
```

Tip: if arms don't need a target value, push a dummy (`null`, `false`, ...) before `switch`; your `case` expressions can be any legal truthy expression.

Loops

Infinite: forever

An empty forever loop just spins. Press **Ctrl-B** (Break) in final or Run/Break in final-gui to stop it.

```
> // ( -- )
: hang
  "This loop runs forever! Press Ctrl-B to Break!\n" $
  forever
/forever
;
hang
```

Iterated: foreach

The `foreach` block is Final's workhorse iterated loop. It can be driven from a variety of compound data structures, such as Lists, Maps, and Objects. You can also use it to easily iterate counted ranges. In all cases, the keyword `i` in the loop refers to the current value of the type being iterated. You can even iterate the Unicode runes in a String this way!

`range` builds an inclusive numeric range. Try a number from 1 to 10 — here `5` prints 1 through `5`, each followed by a space, then a newline after the loop.

```
> // ( end:i -- )
: countTo
  1 swap range
  foreach
    i $ sp
  /foreach
  nl
;
5 countTo

1 2 3 4 5
```

Pretest: begin ... while ... repeat

Condition is tested at the top of every iteration. Zero or more times.

```
> // ( n:i -- )
: drain
  begin
    dup while
      dup $ sp
      1 -
    repeat
      drop nl
  ;
3 drain

3 2 1
```

Post-test: begin ... until

Body runs at least once; the test comes after. Loop continues while the condition is falsy, and exits when it becomes truthy.

```

> // ( n:i -- )
: blastoff
  begin
    dup $ sp
    1 -
    dup 0 == until
  drop nl
;
3 blastoff

3 2 1

```

break and continue

break can be used to exit completely from the innermost enclosing loop among:

- foreach ... /foreach
- forever ... /forever
- begin ... until
- begin ... while ... repeat
- also legacy begin ... again

continue — only in:

- foreach ... /foreach
- forever ... /forever
- (Not on until or while ... repeat.)

Both words can be used in a conditional **if** block—the **break** or **continue** applies to the containing loop.

To percolate either in an outer loop, pass a flag or sentinel from the inner loop, checked by the outer.

Objects and Types

Final provides first-class support for objects, including a concise, expressive syntax and operator set for accessing the members of an object.

All objects in Final are instances of a Type, which is Final's object that holds the definition of an object's members.

An object may have any of the following:

- instance properties, which can be read only
- instance methods, which perform an operation
- class properties, often used for constants
- class methods, often used for instance factories
- ordinal accessors, such as for lists: @() and !()
- name accessors, such as for Maps: @{} !{} and ?{}
- vec and matrix accessors: @[] and ![]

You can get detail on all the properties and accessors provided by a type, by entering `help <typename>` in the UI.

Final's type system is hierarchical, and supports a form of inheritance called prototype inheritance. This is primarily a conceptual concept, that groups members common to a family of concrete instance types. Such types have names like `ProtoList` and `ProtoMap`. It is also used by a number of Final capabilities, such as Json rendering, to provide Final methods that will apply to all the child types. Json uses this to generate Json arrays for all List types, and Json objects for all Map types, even those that have no implementation relation, such as `Map` and `DynamicObject`.

Many of the Types that Final uses and exposes for your use are written in Go, Final's implementation language. But all Objects and Types, whether they are go-native or instances of your defined objects, such as `Dynamic` and `Typed Objects`, share the same means of access and operation, introduced next.

Compound Path Operations

Legacy Forth had a basic policy that all operations be expressed by single, whitespace-separated tokens. During development of Final's Type and Object system, this approach to object operations was found to be lacking in terms of user understanding, and convenience. So a special sub-language of composable object operations was created. These "Compound Path Operations" are a core part of Final,

and used extensively. We cover the basics here, but invite you to dive deeper in the [Objects](#) deep-dive.

The simplest example of an object operation is accessing a readonly constant value of a Type. This convention is used to provide enum values, Type-related values, and math constants. To do this, you simply go `TypeName.property`. Here are common examples:

```
Math.pi // pushes 3.14 etc on DS
Int.maxInt
Float.NaN / pushes the IEEE-754 NaN value
```

Many types provide factory methods for creating a new instance of the type. Here is an example of creating a `DynamicObject`, and then moving it to the Temp Stack for further operations. (Final's extended set of auxilliary stack are covered in-depth in the topic [Stacks](#)). The following sequence shows the steps in creating an instance of a `DynamicObject`, setting some field values, then reading them.

```
DynamicObject.new^ // ( -- do:DynamicObject )
>ts // ( do:DO -- ) t( -- do:DO ) move it to Temp Stack
1 t:.one! // ( 1 -- ) t( do:DO ) creates/sets .one property to 1
t:.one@ // ( -- 1 ) t( do:DO ) fetches the value of .one property
".one is: " $ $ nl // nl prints a newline

.one is: 1
```

Common Path operations:

- moving values to/from auxilliary stacks use eg for Temp Stack, `>ts ts>`
- the “Context Selector”, which specifies on which stack the object resides, uses the `cs:` prefix, where `cs` can be: `t x y z o` If you see `cs:` that means that “`cs`” will have one of those actual values
- Methods are invoked with `.name` then using the `^` affix operator.
- Property values are referenced with `.name`
 - to store a value, the value should be on DS, then use the `!` “store” affix
 - to retrieve a value, which goes to DS, use the `@` (“fetch”) affix
 - readonly/const properties can omit the `@` affix

Due to the logic of how stack-based languages use the Data Stack for the results of operations, it was decided in Final that most compound path operations must start with the object, called the “receiver” (i.e. receives the access commands), on an

auxilliary stack. So a common sequence in Final is creating a new object instance using a factory method, which results in the object on DS, then moving it to an auxilliary stack for all further operations.

Final provides convenience operators for the most common ordinal and by-name operations on objects that support those operations, like lists and maps. Here is a quick summary:

- `cs:@()` // (index:i – value) `cs( obj )`
- `cs:!()` // (value index:i –) `cs( obj )`
- `cs:@{}` // (name:s – value) `cs( obj )`
- `cs:!{}` // (value name:s –) `cs( obj )`
- `cs:?{}` // (name:s – exists:b) `cs( obj )` Matrix operators, examples for 2d mats
- `cs@[ ]` // (row:i col:i – value)
- `cs![ ]` // (value row:i col:i –)

The `help` for every Type lists the accessors that are supported by instances of that type. Usually an object only supports the `()` set or `{}` set, but some, like `DynamicObject`, support both. Mats support the `()` accessors for flat indexed access of the values, and the `[ ]` accessors for eg row/col in 2d Mats.

Paths can be used to walk object hierachies, that include nested instances of other objects. You just use `.propertyname` for the contained instances. The intermediate objects fetched during such a “path walk” are automatically instantiated and elided by the path walker, so never appear on any stack. For example, to find the length of your current dictionary, you walk a path starting with the `DictionaryContext` DC value:

```
dc.dict.len $
```

```
24
```

Indicating you have defined 24 words in your local dictionary so far.

Compound Types

Final provides a useful set of compound objects. These are described briefly below, and covered in detail in the section [Compound Types](#).

Lists

List is Final's dynamically sizable array type. A List starts out empty, after which items can be added, inserted, or deleted. A List can grow to any size.

Lists provide the `@()` and `!()` ordinal accessors for convenient reading/writing of existing elements. The List Type object provides the remaining API for adding, inserting, and deleting.

Lists can be iterated in a `foreach` loop. Just put the List on DS prior to your `foreach` statement, and the iterator is then automatically provided. The `i` value is the current item in the iteration. Iteration is deterministic over the range `0..Len-1`.

List

The type called List is Final's general purpose list. Its items are of type Any, so it can hold any Final basic or object type. Items do not have to be the same type.

StringList

StringList is optimized for the common case of a List of String values. It may only hold values of type String.

Numeric Lists

Final provides a strongly typed, memory-efficient list for all the following numeric types: int8, int16, int32, int64 (aka Int), unsigned 8-bit (**ByteList**), uint16, uint32, uint64 (aka UInt), float32, float64 (aka Float). Values are automatically boxed/unboxed to Any on read/write.

As an example, a **ByteList** (unsigned 8-bit bytes, 0...255) provides a 16x reduction in memory required to store byte-sized values, compared to List. ByteList also finds use in the Final runtime for interoperation with String, whose values are stored in an underlying array of bytes; ByteStreams, that are used for byte read/write of files; and in some Http scenarios.

StringKeyValueList

This specialized list is used in Http classes for defining query parameters, and to hold Http headers, all of which are allowed to have multiple instances of the same key, therefore, Map-like objects cannot be used for these cases. The List stores values of type `StringKeyValue` in which `.Key` and `.Value` are both of type `String`. The list preserves the ordinal order of the source, such as in `HttpRes` headers.

The `i` for `foreach` iteration is a `StringKeyValue` object.

Maps

A map is a collection of key:value pairs. Access is by key. The first write of a key value creates the entry. Subsequent writes to that key, overwrite the previous value.

Maps provide the `@{}` and `!{}` accessors for reading/writing a value by its key. For reads, the key must exist, otherwise an error occurs. The `?{}` accessor can be used to determine if the key exists.

Maps can be iterated. Place the Map object on DS prior to a `foreach`. The order of iteration is indeterminate. `i` is an object of type `KeyValue`, providing the key and the value of the entry.

Final provides 2 Map variants.

Map

Map holds keys and values of type `Any`. The values can be of any type, but the key must be a type that can be compared, such as `String` or `Numbers`. If you try to use an object as a key, an error will occur. Map is provided primarily to support the scenario of using numeric keys.

StringMap

`StringMap` is specialized for the common case of a map whose keys are `Strings`. Only a `String` is accepted as a key. Like `Map`, values can be of any type.

Set

Set holds values of any comparable type. It only contains one instance of any value added.

Sets have operations for producing a new Set from 2 input Sets: union, intersection, difference, and symmetric difference. Sets can be iterated

StringSet

StringSet is specialized for the common case of a set whose values are Strings. .

Vecs and Mats (Vectors and Matrices)

A Vec is a one dimensional ordinal collection of values with a fixed length set at construction time. Final provides VecNf which holds Float values, and can be created with any length. Vec use and its operators are covered in the [Linear Algebra](#) section.

A Mat is a multi-dimensional ordinal collection of values, with fixed dimension sizes set at construction time. Final currently provides a variety of 2-dimensional Mats, addressed using the row/column convention. They are stored in row-major order into an underlying flat array of values. The Mat2f Mat type, in addition to being useful generally, also has special operators and roles in the Linear Algebra/Ordinary Least Squares regression system.

Mats and Vecs both use the `@[]` and `![]` accessors to read and write values. No additional method-based API is provided. For Vecs, you provide one index, For Mats you provide the indexes in order of outer-level to inner-level; for 2-d mats, your provide (row:i col:i – value) for `@[]` and (value row:i col:i –) for `![]` .

Mat2TN

Where T is type, N is bits. A family of numeric Mats that hold signed/unsigned integral or float values, of different sizes. Some, in addition to being useful generally, also have special use in the [Drawing/Blitting](#) framework.

Mat2c

A Mat that stores Color values; also used in Blitting.

Mat

A mat, comparable to List, that can store any Final basic, compound type, or object instance.

DynamicObject

DynamicObject (DO) is a little dynamo of a type, and is used often by the Final runtime, and in Final code. A DO is similar to a Map, but has less overhead, and is faster to read and write than a Map for a number of entries in the range of less than ~64 (however it does not have a size restriction, and automatically optimizes access for field counts greater than 64.). You can also access the fields by name using the dotted `.fieldName` accessor.

You can use it like an object, creating fields dynamically using the `.fieldName!` notation, or like a map, using the `!{}` accessor. A field must be created before it may be read; if a read is attempted on a field that does not exist, a runtime error occurs. Like Map, DO provides the `?{}` accessor.

Unlike Map, DO preserves the order of field entry. When you iterate a DO, it iterates in order of field creation. DO also provides a highly optimized set of ordinal accessors `@()` and `!()` which read and write a value by its ordinal order of creation. This form of access avoids the more expensive field name lookup, when you are certain of the order. Dynamic vectors such as `.x.y.z` for example, can be accessed like an `array[2]` type of object.

DynamicObject instances are one-offs, but they can be (shallow) cloned, or used as a prototype to create instances of StaticObject.

StaticObject

A StaticObject is used much like a DynamicObject. The major difference is that a StaticObject is created from a DynamicObject that serves as its prototype. Once created, a StaticObject has only the fields of the DynamicObject at the time of its creation. You cannot add new fields to a StaticObject.

StaticObject instances use much less memory than a DynamicObject, and are used for things like Database query result sets, and Table rows.

TypedObject

TypedObject is similar to DynamicObject, but its fields are more formally defined, with a schema of `fieldname:Type` pairs. Once defined, a TypedObject becomes its own Type in the Final Type registry, and can be used to create new instances.

A TypedObject may also have methods, written in Final.

Runtime behavior of a TO instance is similar to a DO, including definite iteration in order of field definition and ordinal accessors for faster access.

When you use the `.methodName^` syntax to invoke a Final method on a TO instance, the instance exists on the OS for the duration of the method, providing convenient access to its members via the `o:` prefix accessor. The instance remains on OS for the lifetime of the method, and is removed automatically by the runtime when the method returns.

Table

A Table is a set of rows with a header, similar to a spreadsheet. It supports sorting of the rows by any column using progressive stable sorting, which preserves the order of the previous sort-step for ties.

A Table can be made from a `DynamicObject`, which then becomes a prototype for `StaticObject` rows. It can be created from a `TypedObject`, for instances of that type. It can be created from a `Type` that has fields, in which case, the fields become the column names. It can also be used manually by supplying a list of column names, and then adding any compatible objects as rows. A “compatible object” is any object that provides named or ordinal field access.

The `dbIter>table` word efficiently creates a table from the result set of a Database query.

Features

This entry contains a whirlwind list of Final’s other features. Each it explained in detail in its own entry. Now that you’ve had a basic introduction to Final, you are free to browse around. However it is recommended to first read the topics in the Foundations section, to get a good grounding of Final’s core constructs and operations.

- efficient OpCode-based VM with fast jump-table dispatch
- full support for Types and Objects
- concise, expressive “compound path” syntax for object operations
- auxilliary stacks, for convenient object parking and access
- support for Iterators, and convenient `foreach` iterated loop
- excellent assortment of often-used collection types
- `{ ... }` and `sm{}` builders for DynamicObject and StringMap; DynamicObject `.clone^` for prototype-style copies; StaticObject via `do.newStatic^` for fixed-schema rows
- TypedObject class constants plus Final methods via `: Type#method ... ;` (instance) and `: Type##method ... ;` (class)
- Table — columnar collection with schema from DynamicObject / TypedObject / Types; StaticObject rows; stable `sortByOrd` / `sortByName`
- full support for Unicode strings as Utf-8, including efficient Rune iteration; wide variety of `str::` operations
- DateTime and Interval support, with formatting via `dt::fmt` (plus constants such as `DateTime.iso`, `Interval.clock`)
- expressive and convenient `()` operators for creating value list to `fmt` operator
- support for a variety of variants of TextStream and ByteStream interfaces:
 - files
 - strings (reading) StringBuilder (writing)
 - stdio
- File system — `File`, `FileInfo`, `Dir`, `Path`; metadata, copy/move/delete, directory iteration
- Regex, including compiled regex, matches, selectors
- full Json rendering and parsing
- Threading, Pipes, and Atomic threadsafe objects
- Web Dial (Http GET/POST) and Web Listen (`HttpListener` — ServeMux routes, path params, REST methods, Pip-per-route handlers)

- SQL databases via `DbProvider` (Postgres, MySQL, SQLite); iterable result rows as `DynamicObjects`; `iter>list`
- Env and Moniker — `cwd` `.env` + process environment; opaque DSN / secret handles with `$KEYNAME` substitution
- Cryptography — Hash (MD5 checksum, SHA-256), Hex/Base64, `SecureRandom`, Key, Cipher (AES-256-GCM / ChaCha20-Poly1305), Ed25519 / X25519, Mac (HMAC-SHA-256), Kdf (HKDF-SHA-256, Argon2id)
- Linear Algebra: `VecNf` / `Mat2f`, `vec::` / `mat::`, OLS `fit` / `compare` / `predict`
- `try` / `catch` / `finally` / `throw` — recoverable exceptions (invalid JSON `throw`s rather than panicking)
- Modules (local `install` / `import`) and `.fnlp` projects (`loadProject`)
- Session configuration (`sess.config`) and pluggable logging (`logInfo`, ...)
- `forget`: reload marks — inclusive wipe back to a mark word for load → try → reload without process relaunch
- Security Restrictions, to restrict operations like file access, web dialing, listening, databases, crypto, threading, and module install/import, for embedded/AI scenarios
- `LockSession` — exclusive Send hold so an agent turn is not interleaved
- full example of a Web-based chat-like app that provides Final as a tool for the Agent
- `finalembd` / `finalgo` Go packages for embedding Final in other Go programs, with Session and Send semantics
- VS Code / Cursor Final extension for `.fnl` — TextMate highlighting, 2-space indent, sandwich snippets (local VSIX / F5 install)
- Raspberry Pi 5 support, for writing Pi-based control systems.
- Command line support, OS pipes, shebang scripts (Unix-like only)
- Raster blitting and drawing operations (all Final versions)
- GUI support via Fyne controls, `final-gui` binary
- support for Games and Simulations via `SimGame` framework
- Sound support via `SoundPlayer` and `Sound` objects

Near-term Features Planned

- Language-server features for the editor (dictionary completion, stack-effect hover, diagnostics)
 - build on compiler execution state tracking
- Compiler execution state tracking
 - catch stack bugs: underflow/leak
 - find opportunities to use strongly typed axioms rather than polys

- Markdown parsing
- Html and Xml rendering
- Interactive Web tutorials, using a sandboxed Final Session

Stacks

Up to now, we have been mostly mentioning the Data Stack (DS). But Final also defines four auxiliary stacks, all of which have the same capabilities, and a special Object Stack (OS) that contains the object instance when a Final method runs. The stacks, stack operators, and context selector prefixes are described in the following entries.

Stack Summary

Programmer accessible runtime stacks are a defining feature of stack-based languages like Forth and Final.

Final has seven(!) stacks, defined below. Note that this may seem to be a *LOT* of stacks, in comparison to Forth, which only had *rs* and *ds*. But the needs of a modern language, which will typically involve the use of several objects in an operation, along with the many shortcomings and dangers of Forth's need for complex *ds* juggling have led to this design.

Return Stack (*rs*)

The Final Return Stack is similar to the singular “stack” in conventional languages and machine code. It's primary purpose is to hold return addresses when a nested defined word runs. It also holds some additional values, such as a ContextMark containing the depths of all other stacks, and Dictionary-related lengths for some operations. The foreach iterated loop stores the iterator on *rs*.

The Return Stack is not user accessible in Final; it is used solely by the runtime.

Data Stack (*ds*)

This is the core stack used for holding values during an operation, and is almost directly comparable to Forth's “stack”. Unlike procedure-based languages, parameters to, and results from most low and high level operations in Final are done via *DS*. This enables a system of operation decomposition and composition which provides numerous advantages over the procedural approach, in which a simple factoring for readability, testability, or reuse, requires a re-expression of all the parameters in the function call and return. In Final, you just factor, and the stacks hold what they have at the factor point.

Final allows compound operations, such as property and method access on objects, indexed accessors on Lists, and named accessors on Maps, to originate from objects on any stacks. But the convention for all those operations is for the parameters and results to be located on *DS*. *DS* thus serves as a common “hub”, for the source of parameters, and target for results of most operations.

Auxilliary Stacks (**ts**, **xs**, **ys**, **zs**)

Final provides four additional stacks, which all have the same capabilities. These can be used to help stage source and destination objects during a sequence of operations, without choking ds with multiple values that need to be constantly juggled. Doing this in Forth required constant juggling of ds via sundry dups drops overs nips tucks rots etc. which are difficult to reason about and get correct while writing, and almost impossible to track when trying to read code.

In addition to their general purpose uses, `ts`, the “Temp Stack”, is, by convention, the stack for short-lived parking: compound handles during `t:`, and interpret/compile temps. Prefer `ts` over the other stacks for any word-local intermediates.

Guidelines for effective use of the auxilliary stacks is provided in [Writing Clean Final Code](#)

Object Stack (**os**)

The Object Stack (os) is used exclusively by the runtime, to push the instance of an object for a Final Method call. Therefore, the runtime of such methods always access their instance via `os.tos`; and the appearance of the instance on a stack during the call doesn’t interfere with whatever are the context of objects and values already existing on the other stacks. (Note: Class methods push the Type on os, not an instance. Thus, other class methods and properties are accessible in that calling context.)

The Object stack can use the instance as a root for all OO accessor operations, by prefixing the operation or sequence with `o:` as with `txyz`.

However unlike the other stacks, you may not perform stack manipulation operations on os (e.g. `o:drop`, `o:dup` etc) nor may you push values onto os; there is no `>os` and `os>` operator pair. The runtime takes care of pushing the inst value to os before the method dispatch, and popping it after the method finishes. Note that this mechanism fully enables nesting Final method calls; the latest inst is always the one on os when that method runs.

Context Selectors

Given that Final has six user-accessible stacks, a means is obviously needed for operations to specify to which stack you are referring. This is called Context Selection. It arises in two areas:

1. Many stack operations can be performed on any stack, so in this case, a Context Selector (cs) specifies which stack the operation should be performed on.
2. Compound Path operations can start with an object—the “receiver”—on most any stack, so the cs is required to specify which stack.

A cs is simply the stack letter, followed by a colon, eg

- t: prefix for Temp Stack
- x: prefix for X Stack
etc.
- o: prefix for O Stack

The Data Stack does not have a cs selector; an operation that supports a CS, without one, is assumed to apply to DS.

Note that not all compound path operations are supported with the receiver on DS; most require the receiver to be located on an auxilliary or the O Stack.

Stack Operators

Final has two kinds of stack operators:

- explicit stack selection
- Context Selector (cs) stack selection

As explained in the previous section, cs-aware operators referring to the Data Stack do not use a selector, DS is the default.

Some of these operators have been inherited from Forth, which as explained earlier, often required programmers to engage in mind-numbing stack value juggling; Final's design and coding conventions have greatly reduced the need for such juggling, so don't worry about words like `rot` or `pick` and such—they don't find much use in Final. `dup` `drop` `peek` and the interstack movement operations make up the vast majority of Final's stack repertoire.

The term `tos` means Top of Stack.

Explicit Stack Selection Operators

These operators explicitly state the stacks involved. All the to/from ones imply DS as the unspecified half. These operators are used frequently.

Op	Effect	Description
<code>>ts</code>	<code>(val:a -) t(- val)</code>	move value from DS to TS
<code>ts></code>	<code>(- val:a) t(val -)</code>	move value from TS to DS
<code>tdrop</code>	<code>t(val:a -)</code>	discard tos of TS
<code>>xs</code>	<code>(val:a -) x(- val)</code>	move value from DS to XS
<code>xs></code>	<code>(- val:a) x(val -)</code>	move value from XS to DS
<code>xdrop</code>	<code>x(val:a -)</code>	discard tos of XS
<code>>ys</code>	<code>(val:a -) y(- val)</code>	move value from DS to YS
<code>ys></code>	<code>(- val:a) y(val -)</code>	move value from YS to DS
<code>ydrop</code>	<code>y(val:a -)</code>	discard tos of YS
<code>>zs</code>	<code>(val:a -) z(- val)</code>	move value from DS to ZS
<code>zs></code>	<code>(- val:a) z(val -)</code>	move value from ZS to DS
<code>zdrop</code>	<code>z(val:a -)</code>	discard tos of ZS

CS-Aware Stack Operators

These operators act on the stack selected by a Context Selector (`t:` , `x:` , `y:` , `z:` , or none for DS).

Op	Effect	Description
<code>dup</code>	<code>cs(val:a – val val)</code>	duplicate the value on tos
<code>dup2</code>	<code>cs(a:a b:a – a b a b)</code>	duplicate the top two values
<code>drop</code>	<code>cs(val:a –)</code>	discard tos
<code>drop2</code>	<code>cs(a:a b:a –)</code>	discard the top two values
<code>swap</code>	<code>cs(a:a b:a – b a)</code>	swap the top two values
<code>over</code>	<code>cs(a:a b:a – a b a)</code>	copy the second value to tos
<code>nip</code>	<code>cs(a:a b:a – b)</code>	discard the second value
<code>tuck</code>	<code>cs(a:a b:a – b a b)</code>	copy tos under the second value
<code>pick</code>	<code>cs(... +n:i – ... val)</code>	copy the nth value to tos (0 = tos)
<code>rot</code>	<code>cs(a:a b:a c:a – b c a)</code>	rotate the top three values
<code>peek</code>	<code>cs(val:a – val) (– val)</code>	copy tos of CS onto DS
<code>ds></code>	<code>cs(val:a –) (– val)</code>	move tos of CS onto DS
<code>>ds</code>	<code>(val:a –) cs(– val)</code>	move tos of DS onto CS
<code>depth</code>	<code>cs(– n:i)</code>	depth of the selected stack

Control Flow

The [Quick Tour](#) already introduces Final's control-flow blocks (`if` / `switch` , loops, `break` / `continue`) and the context-effect comment convention.

This section goes deeper on a few of those items:

Page	Focus
range and rangeby	Building counted iterators for <code>foreach</code>
Iterators: foreach	Which types iterate, and what <code>i</code> is in each case
Other Loops	Non- <code>foreach</code> loops: <code>forever</code> , <code>begin ... while ... repeat</code> , <code>begin ... until</code> , legacy <code>begin ... again</code> , plus <code>break</code> / <code>continue</code>
Subwords	Subwords are “word-lite” words defined within a colon word

All of these words are **compile-only**: they belong inside a colon definition (`:` ... `;`). Trying them at the interpreter prompt raises an error.

`try` / `catch` / `finally` / `throw` are also compile-only; they are covered in [Error and Exception Handling](#).

For stack effects and short descriptions, use `help` on the word name (for example `help foreach` , `help forever`).

range and rangeby

Final provides two useful words for setting up counted iterated loops. `range` is for a definite counted loop with an increment of 1. `rangeby` is more flexible. They are described below. For detailed descriptions and stack effects, use `help`.

range

`range` takes an inclusive `Int` start and end value, and can then provide an `Iterator` for `foreach`. `i` is the value of the iteration value for that step, and starts with `beg` and ends with `end`. When using `range` to iterate over a counted zero-based range, you set `beg` to 0 and the end to be `count-1`. If the count is 0, no iterations are performed. The resulting `i` will be an `Int`.

```
> // ( -- )
: count1to5
  1 5 range
  foreach
    i $ sp
  /foreach
  nl
;
count1to5

1 2 3 4 5
```

Zero-based index style (here: three slots, indices `0` `1` `2`):

```
> // ( count:i -- )
: indices
  0 swap 1 - range
  foreach
    i $ sp
  /foreach
  nl
;
3 indices

0 1 2
```

An empty range is legal and runs zero iterations — useful when a computed end would otherwise need a special “nothing to do” case.

rangeby

`rangeby` takes a beg, end, and step value. Step can be negative if beg>=end. The values can be of any numeric type, but they must all be the same type. This enables fractional Float step values. The iteration stops when the value goes beyond the end value. As with range, an empty range that has no first value is legal, and results in no iterations. `i` is the same type as the inputs.

```
> // ( -- )
: byTens
  0 40 10 rangeby
  foreach
    i $ sp
  /foreach
  nl
;
byTens

0 10 20 30 40
```

Float step:

```
> // ( -- )
: halves
  0.0 2.0 0.5 rangeby
  foreach
    i $ sp
  /foreach
  nl
;
halves

0 0.5 1 1.5 2
```

Counting down (negative step; `beg >= end`):

```
> // ( -- )
: countdown
  5 1 -1 rangeby
  foreach
    i $ sp
  /foreach
  nl
;
countdown

5 4 3 2 1
```


Iterators: foreach

Final supports the modern concept of Iterators. An iterator is an object produced from a source object, that knows how to iterate over values of the source. Many Final Types can provide an iterator, and their values can thus conveniently be iterated over in a `foreach` block.

To use them, you simply put the instance on DS prior to the `foreach`; `foreach` automatically retrieves the iterator, and starts the loop (if the iterator is not empty, eg an empty List). You access the item with the keyword `i`; in each case `i` is defined by the object, and is typically the natural value of the container.

You can nest `foreach` loops. Each level accesses its items via `i`. You can access the items above your level via the `j` keyword (next level up) and `k` (second level up.) Nested `foreach` loops use `i`, then `j`, then `k` for the current item at each nesting level.

The `i/j/k` accessors are only defined within the static lexical nesting of `foreach` loops. They are only defined within the word that has declared the loop; you cannot call another Final word, and use them in that context. You can however freely use them in subwords, provided they exist at the time the subword accesses them; i.e. you are calling the subword from within a `foreach` where `i/j/k` (as applicable) exist.

The following types support iteration.

range and rangeby

As described in [Ranges](#).

```
> // ( end:i -- )
: countTo
  1 swap range
  foreach
    i $ sp
  /foreach
  nl
;
5 countTo

1 2 3 4 5
```

Lists: List, StringList, all Numeric Lists, StringKeyValueList

- Most Lists: iterate over the values; `i` is the list item; the list is iterated in 0-based order
- StringKeyValueList: `i` is a StringKeyValue object (Key and Value are strings); this List is principally used for Web query parameters and headers; thus, unlike Maps, keys can be repeated.

```
> // ( -- )
: showList
  ( "alpha" "beta" "gamma" )
  foreach
    i $ sp
  /foreach
  nl
;
showList

alpha beta gamma
```

StringKeyValueList — print each pair (`.add^` takes value then key):

```
> // ( -- )
: showPairs
  StringKeyValueList.new^ >ts
  "1" "a" t:.add^
  "2" "b" t:.add^
  ts>
  foreach
    i >ys
    y:.key@ $ "->" $ y:.value@ $ sp
    y:drop
  /foreach
  nl
;
showPairs

a->1 b->2
```

Map family: Map, StringMap, DynamicObject, StaticObject, TypedObject

- Map, StringMap: iterates over all entries in an indeterminate order; `i` is a KeyValue object

- `DynamicObject`, `StaticObject`, `TypedObject`: iterates deterministically in order of field definition; `i` is a `KeyValue` object

```
> // ( -- )
: showDO
  DynamicObject.new^ >ts
  10 t:.x!
  20 t:.y!
  ts>
  foreach
    i >ys
    y:.key@ $ "=" $ y:.value@ $ sp
    y:drop
  /foreach
  nl
;
showDO

x=10 y=20
```

Creation order is preserved for `DynamicObject` / `TypedObject`; `Map` / `StringMap` order is not defined — do not rely on it.

Sets: `Set`, `StringSet`

`foreach` yields each **value** (not a `KeyValue`). Order is not defined.

```
( 1 2 3 set)
foreach
  i $ sp
/foreach
nl
```

`VecNf` and `Mat2*`

`foreach` yields each value. A vector walks `0 ... len-1`. A matrix walks **flat** row-major cells. For row/column coordinates see [Vectors and Matrices](#).

Mats provide a convenient `getRowIter` and `getColIter` that are basically ranges set to the proper values for the row and column size respectively.

Pipes

Pipes provide a special purpose iterator that enables you to block waiting for the next value to be received, get that value via `i`, then resume checking for the next value. When the pipe is closed by the producer, the `foreach` exits. see [Threading](#).

Other Loops

The [Quick Tour](#) shows `foreach` for iteration. Final also has indefinite loops that are **not** driven by an iterator. This page covers those, plus `break` and `continue`.

Like all control-flow blocks, these words are **compile-only** — use them inside `: ... ;`.

Form	Kind	Notes
<code>forever ...</code> <code>/forever</code>	infinite	Prefer this over legacy <code>begin ... again</code> ; supports <code>continue</code>
<code>begin ... while ...</code> <code>repeat</code>	pretest	Body runs zero or more times
<code>begin ... until</code>	post-test	Body runs at least once ; exit when condition is truthy
<code>begin ... again</code>	legacy infinite	Prefer <code>forever</code> ; no <code>continue</code>

Infinite: `forever ... /forever`

No parameters. Loop until `break`, a fault, or an interactive Break (Ctrl-B in final).

```
> // ( n:i -- )
: tickN
  forever
    dup 0 <= if drop break then
    dup $ sp
    1 -
  /forever
  nl
;
3 tickN

3 2 1
```

An empty `forever ... /forever` spins forever — use Ctrl-B to stop it in the TUI.

Pretest: `begin ... while ... repeat`

Words between `begin` and `while` run each attempt; `while` consumes a Truthy flag. Truthy → run the body through `repeat` and branch back to `begin`. Falsy → exit after `while` (skip the body).

```

> // ( n:i -- )
: drain
  begin
    dup while
      dup $ sp
      1 -
    repeat
      drop nl
  ;
3 drain

3 2 1

```

If the first test is already falsy, the body never runs:

```

> // ( -- )
: never
  0
  begin
    dup while
      "should not print\n" $
    repeat
      drop
  ;
never

```

Post-test: begin ... until

The body runs, then `until` consumes a Truthy flag. **Falsy** → **loop back** to `begin`; **truthy** → **exit**. So the body always runs at least once.

```

> // ( n:i -- )
: blastoff
  begin
    dup $ sp
    1 -
    dup 0 == until
  drop nl
;
3 blastoff

3 2 1

```

Legacy infinite: begin ... again

Same idea as `forever`, but older spelling. Prefer `forever ... /forever` in new code — `again` does **not** support `continue`.

```
> // ( n:i -- )
: tickAgain
  begin
    dup 0 <= if drop break then
    dup $ sp
    1 -
  again
  nl
;
3 tickAgain

3 2 1
```

break

Exit the **innermost** enclosing loop among:

- `foreach ... /foreach`
- `forever ... /forever`
- `begin ... until`
- `begin ... while ... repeat`
- legacy `begin ... again`

Typical pattern: test inside the loop, then `if break then`.

```
> // ( -- )
: findThree
  1 10 range
  foreach
    i 3 == if break then
    i $ sp
  /foreach
  nl
;
findThree

1 2
```

When `i` is `3`, `break` exits before the print, so only `1 2` appear.

continue

Skip the rest of the **current** iteration. Allowed only in:

- `foreach ... /foreach`
- `forever ... /forever`

Not in `until` or `while ... repeat`.

```
> // ( -- )
: oddsOnly
  1 6 range
  foreach
    i 2 % 0 == if continue then
    i $ sp
  /foreach
  nl
;
oddsOnly

1 3 5
```

Inside `forever`, `continue` jumps back to the top of the forever body (here: skip printing `2`):

```
> // ( -- )
: demoSkip
  3
  forever
    dup 0 < if drop break then
    dup 2 == if 1 - continue then
    dup $ sp
    1 -
  /forever
  nl
;
demoSkip

3 1 0
```

Both `break` and `continue` may sit inside an `if / else / then` — they still apply to the **containing loop**, not to the conditional.

To affect an **outer** loop, set a flag or sentinel in the inner loop and test it in the outer loop after `/foreach` or the matching closer.

Subwords

Final lends itself well to factoring complicated words into smaller worker words. This can be done at a macro level by defining new colon words, and then delegating to them in a main word.

But such factoring can be inconvenient or have drawbacks for several reasons:

- Final has a global namespace, so mnemonic words like “init” are not good factored word names;
- a delegated colon word doesn’t inherit locals set up in the calling word;
- when reading a file, it can be difficult to determine if a word is called in more than one place, or is just a factor word;
- calling a colon word has overhead, which makes them less desirable for hot inner loop steps.

To address these issues, Final has subwords. A `subword` is a word defined inside a main colon word. Subwords are much lighter and more efficient than colon words, so are well suited for factoring code in inner loops or switch cases, to make the main flow more readable.

Here is an example use:

```
: main-word
:s init
  // some factored init code, for clarity
;s

// ( item -- )
:s do-item
  // code to handle one item in a loop
;s

init
myList
foreach
  i do-item
/foreach
;
```

A subword has access to all the context of the main word, especially the local variables, and the *i/j/k* variables of `foreach` loops, and the *a/b/c* variables of `switch` statements. (And of course the stack contexts, which factored colon words also have

access to.) Unlike a factored colon word, a subword may compile `i / j / k / a / b / c` without those blocks lexically around the use — they are checked at runtime against the shared frame. In the main body of the word, the usual static nesting rules still apply.

Subword definitions must be the first elements in a main word. There can be no main word code until the last subword definition. Subwords themselves cannot be nested; no `:s` in an outer `:s`.

You use subwords exactly like any colon word in your code, as demonstrated in the examples.

The subword names are only visible inside the main word, not outside it. You cannot invoke a subword inside one word from another. The same ordering rules that apply to words apply to subwords: a subword may call another subword that is lexically defined before it, but not after.

Subwords cannot call themselves. The `recurse` statement can be used, but it causes a new invocation of the main word, not the subword.

You can use most of the structured control blocks in a subword, such as `if/then`, `foreach`, etc. But the following restrictions apply:

- you may not define a `try/catch` block (but you can use `throw`)
- if you define a `foreach` loop, then it will nest inside any active outer `foreach`; you can access those using `j/k` but only if they are active when you call the subword
- the same rule applies for `switch` and `b/c/` accessors

Most importantly, you can access all the locals defined in the main word, or subwords, within a subword. A subword can also add locals. Since local variables are resolved when they are accessed, not when they are compiled, it doesn't matter if they are defined lexically later than where they are used. That said, for clarity, you may choose to initialize the locals that will be used throughout the words, in a subword like the `init` I gave in the example—this will make scanning the subwords prior to reading the main code more clear.

Subword invocation is extremely efficient. A subword is basically like a machine language subroutine, calling one is just a short `brsub` axiom, and returning is a `retsub` axiom, using a fast simple return stack in the `ColonFrameMark` of the main word. Using one will add negligible overhead even to hot inner loops, due to CPU

caching. Therefore, subword factoring is encouraged even for hot paths, if it makes the code clearer.

Objects

The [Quick Tour](#) introduced Types, members, and the basics of compound path access. This Foundations section goes deeper on the **APIs** and the **path syntax** you use every day.

Everything you reach through a path—Go-native types, `DynamicObject` / `StaticObject`, `TypedObject`, `Words`, collections—uses the same API:

Kind	Role
Type	Catalog entry that describes members; CapCase dictionary word (<code>List</code> , <code>Int</code> , ...)
Instance	Concrete value of that Type
Property	Named readable/writable (or const) member — <code>@</code> / <code>!</code>
Method	Named operation — invoke with <code>^</code>
Collection accessors	Indexed <code>@()</code> / <code>!()</code> , keyed <code>@{}</code> / <code>!{}</code> / <code>?{}</code> , coordinate <code>@[]</code> / <code>![]</code> (<code>Mat2f</code>)

Parameters and results for member ops still use **DS**. The **receiver** (the object the path acts on) usually lives on an auxilliary stack (`t:` / `x:` / `y:` / `z:`) or on **OS** (`o:`) inside a Final method. See [Stacks](#) and [Writing Clean Final Code](#).

Concrete collection APIs (`List`, `Map`, `DynamicObject`, `StaticObject`, `TypedObject`, `Table`) are in [Collection Types](#). `DynamicObject` also offers `.clone^` (shallow prototype copies) and `.newStatic^` (mint a [StaticObject](#) from a DO prototype). Use `help TypeName` in the TUI for the full member list of any type.

In this section

- [Types](#) — Type objects, class vs instance members, factories
- [Properties](#) — fetch, store, const, refs
- [Methods](#) — instance and class methods, `^` , Final methods on OS
- [Collection Accessors](#) — `@()` `!()` `@[]` `![]` `@{}` `!{}` `?{}`
- [Compound Path Syntax](#) — path elements, stacking rules, nested walks, examples

Types

A **Type** is Final's description of a value family: instances of which can have properties, methods, and collection accessors. A Type may also have class properties and class methods, both accessed through the Type itself.

Types are first-class objects. CapCase names such as `Int`, `List`, and `DynamicObject` are dictionary words that push the Type onto DS.

```
List $           // prints the Type
List.new^ $      // factory: empty List on DS
```

Primitives (`Int`, `String`, `Bool`, ...) do not have object *instances*, but they still have a Type with useful **class** members (constants, conversions, factories). Reference types (`List`, `Map`, `Word`, ...) have both class members and instance members.

Class vs instance

Receiver	What you can call	Examples
The Type itself	class properties, class methods	<code>Math.pi</code> , <code>Int.maxInt</code> , <code>List.new^</code>
An instance	instance properties, instance methods, items accessors	<code>t:.len@</code> , <code>t:.append^</code> , <code>t:@()</code>

Class methods are never invoked from an instance (`myList.new` is invalid). Factories live on the Type: `List.new^`, `Person.new^`.

By convention:

- default factory is `new` — (`-- inst`) with default values or zeros in the properties
- parameterized factories use names like `newWithCapacity`, `newFromString`, etc.

```
16 List.newWithCapacity^ >ts
"Ada" StringBuilder.new^ >xs
```

Exploring types

```
help List
help DynamicObject
help Word
```

`help` lists properties, methods, class members, and which item accessors the type supports.

Prototype families

Final groups related concrete types under prototype parents such as **ProtoList** and **ProtoMap**. That is mainly a catalog / sharing story (for example Json rendering treats all list-like and map-like types uniformly). Day-to-day code still names the concrete type (`List`, `StringMap`, `DynamicObject`).

Your own types

`TypedObject` registers a named schema in the catalog. After `makeType`, the new CapCase name behaves like any other Type: `YourType.new^`, dotted fields, optional Final methods.

Properties

A **property** is a named member of a Type accessed with `.name` plus a fetch or store affix. Backing may be a stored slot or a computed getter/setter; the API is the same.

Property names and instance method names on one Type are disjoint. Use `help TypeName` to see which is which.

Fetch and store

Form	Effect	Notes
<code>cs:.name</code> or <code>cs:.name@</code>	<code>(-- value) cs(obj - obj)</code>	read; bare <code>.name</code> implies <code>@</code>
<code>cs:.name!</code>	<code>(value --) cs(obj - obj)</code>	write
<code>TypeName.constProp</code>	<code>(-- value)</code>	class const; Type receiver is elided

Value in / value out always on **DS**. The rooted receiver stays on the context stack.

```
DynamicObject.new^ >ts
"Ada" t:.name!
36 t:.age!
t:.name@ $ sp t:.age@ $ nl    // Ada 36
t:drop
```

Store paths onto a parked receiver need an explicit context selector (`t:` / `x:` / ...).

A bare store on DS without a prefix is rejected (`d:` does not help). Ident-rooted paths are allowed without a selector (`1 sess.config.paramA!`), as are `1.field!` and `pls.field!`.

Readonly / const properties (including many class constants) may omit `@`:

```
Math.pi $
Int.maxInt $
Float.NaN $
TokenType.TokenEOF $
```

Const vs ordinary

- **Const / IsConst** — fetch returns the value immediately; there is no lasting PropertyRef. Typical for math constants and enum-like class properties.
- **Ordinary** — fused `.name@` / `.name!` are one-shot (no ref left on the stack). Mid-path `.name` also fetches the value and continues the walk with that value as the next focus.

Property refs (~)

When you need the ref itself (for example read-modify-write with separate `@` / `!`), materialize with the `~` affix. `~` must be the **last** segment of that path token.

```
// Word on DS; materialize name ref, then Forth-style R-M-W
"dup" find
.name~
dup @           // fetch current name
drop
"fresh" swap ! // store; consumes ref
```

Standalone `@` and `!` consume a PropertyRef on the context stack. Fused `.name@` / `.name!` never leave a ref behind.

You cannot fuse `~` with `@` in one token (`name~@` is illegal). Use `.name~` then a separate `@` or `!`.

Nested property walks

Intermediate `.prop` segments fetch the contained object and continue; intermediates are not parked on a stack:

```
dc.dict.len $      // DictionaryContext → dict → len
```

See [Compound Path Syntax](#) for the full segment vocabulary.

Methods

A **method** is a named operation on a **Type**. Invoke it with the `^` (“run”) affix. Bare `.method` without `^` is an error.

Operands and results use **DS**. The receiver stays on the context stack for rooted calls.

Instance methods

```
List.new^ >ts
10 t:.append^
20 t:.append^
t:.len@ $ nl      // 2
t:.clear^
t:drop
```

Form	Effect
<code>cs:.name^</code>	invoke instance method; fused one-shot
<code>.name~ then ^</code>	materialize MethodRef, then invoke (ref consumed)

```
StringBuilder.new^ >ts
"hi" t:.writeString^
t:.toString^ $ nl    // hi
t:drop
```

Class methods

Class methods run with the **Type** as receiver. Factories are the common case:

```
List.new^                // ( -- list )
16 List.newWithCapacity^ // ( cap:i -- list )
5 List.newWithLen^        // ( len:i -- list )
```

Other class methods query or compute without an instance handle:

```
Float.NaN Float.isNaN^ $ nl    // true
2 10 Float.pow^ $ nl           // 1024.0
```

For string→number conversion, use the cast operators from the Quick Tour (`"42">>Int?`), not a Type method.

Final methods (TypedObject)

Methods written in Final are defined on a TypedObject type (see [TypedObject](#)) with `: Type#name ... ;` (instance) or `: Type##name ... ;` (class). Registration happens at `;` — no dict word is left behind. While the method runs, the receiver is on the **Object Stack**. Use the `o:` prefix; you may not push or drop OS yourself; the runtime takes care of pushing the instance prior to invoking the method, and then dropping it when the method returns.

```
: Person#describe
  o:.name@
  o:.age@
  "%s (%d)" fmt
;

Person.new^ >ts
"Ada" t:.name!
36 t:.age!
t:.describe^ $ n\    // Ada (36)
t:drop
```

Nested `o:...` paths behave like any other rooted compound path for the duration of the call.

Collection Accessors

Some Types expose **item** access in addition to named properties and methods. Items are not properties: they use the fused collection affixes below. Index or key operands are on **DS**; with a context selector (`t:@()` , ...) the collection **receiver stays** on that stack.

Affix	Effect	Typical types
<code>cs:@()</code>	(index:i -- value)	List family, DynamicObject / TypedObject by ordinal; <code>VecNf</code> / <code>Mat2f</code> flat
<code>cs:!()</code>	(value index:i --)	same; writes an existing slot
<code>cs:@[]</code>	(i0:i i1:i ... -- value)	<code>Mat2f</code> : (r:i c:i -- f)
<code>cs:![]</code>	(value i0:i i1:i ... --)	<code>Mat2f</code> : (f r:i c:i --)
<code>cs:@{ }</code>	(key -- value)	Map family, DynamicObject, TypedObject; missing key errors
<code>cs:![{ }</code>	(value key --)	create or replace
<code>cs:?[{ }</code>	(key -- exists:b)	Map family, DynamicObject, TypedObject

`help TypeName` shows which of `@()` / `!( )` / `@[]` / `![ ]` / `@{ }` / `![{ }` / `?{ }` that type supports. Most types support either the `()` set or the `{ }` set; DynamicObject and TypedObject support both. `Mat2f` supports flat `()` and coordinate `[]`.

Ordinal `@()` / `!( )`

Zero-based index. `!( )` does not grow a list — use `.append^` / `.insertAt^` for that.

```
List.new^ >ts
10 t:.append^
20 t:.append^
30 t:.append^
1 t:@() $ nl           // 20
99 1 t:!( )           // replace index 1
1 t:@() $ nl           // 99
t:drop
```

On `DynamicObject` / `TypedObject`, `@()` / `!()` use **field creation (or declaration) order**, not a hash order:

```
DynamicObject.new^ >ts
1 t:.x!
2 t:.y!
0 t:@() $ nl          // 1
t:drop
```

On `Mat2f`, `@()` / `!()` are **flat** row-major. Use `@[]` / `![]` for `(row, col)` — [Vectors and Matrices](#).

Mat `@[]` / `![]`

These accessors take a row/col pair, and return or set the value in that cell.

```
2 3 Mat2f.new^ >ts
// set row 0, col 2 to a value
1.0 0 2 t:![]
// retrieve the value
0 2 t:@[] $ // 1
tdrop
```

When iterating in nested `foreach` loops, the following pattern is common:

```
( mat -- )
>ts
t:.getRowIter^
foreach
  t:.getColIter^
  foreach
    j i t:@[]
    // do something with the value
  /foreach
/foreach
tdrop
```

Named @{} / !{} / ?{}

```
StringMap.new^ >ts
100 "count" t:!{}
"Ada" "name" t:!{}
"name" t:@{} $ nl
"missing" t:?{} $ nl // false
t:drop
```

@{} on a missing key is an error. Probe first with ?{} (or a type's `hasKey` method when provided).

```
"city" t:?{} if
  "city" t:@{} $ nl
else
  "n/a" $ nl
then
```

Prefixed vs bare

Day-to-day code should park the collection and use a selector:

```
0 t:@()
"k" t:@{}
```

A bare @() / @{} path whose receiver is on **DS** (no t: / ...) is a one-shot form that **elides** the receiver after the access. Prefer the rooted `cs:` form so the handle remains for the next operation.

Chaining after a get

Because @() / @{} leave a **value** on DS, further member access on that value usually means moving it to a stack first:

```
"tags" t:@{} >xs
0 x:@() $ nl
x:drop
```

Or nest through properties when the path stays inside one walk — see [Compound Path Syntax](#).

Collection type APIs (append, clear, iteration, ...) are documented under [Collection Types](#).

Compound Path Syntax

A **compound path** is a single lexical token that chains object operations with **no whitespace**. It is Final's compact notation for property access, method invoke, and collection get/set. Paths compile and interpret as a left-to-right walk of fixed **segments**.

The Quick Tour covered the everyday forms (`t:.name@`, `List.new^`, `t:@()`). This page enumerates the **path elements**, stacking rules, and longer examples.

Why paths

- **Clarity** — one token states the whole access intent
- **Brevity** — nested objects need not be parked mid-walk
- **Efficiency** — the walker materializes and elides intermediates for you

Rooted paths **preserve** the root receiver on the context stack, so you do not `dup` before each member op. That is the pattern recommended in [Writing Clean Final Code](#).

Path shape

```
[ cs: ] [ RootIdent ] { Segment }...
```

Examples:

Path	Meaning
<code>Math.pi</code>	class const on Type <code>Math</code>
<code>List.new^</code>	class method <code>new</code> , invoke
<code>t:.len@</code>	fetch <code>.len</code> on TS TOS
<code>t:.append^</code>	invoke <code>.append</code> on TS TOS
<code>1 t:!()</code>	store DS value at index into TS list
<code>dc.dict.len</code>	nested property walk from <code>dc</code>
<code>t:.name~</code>	materialize PropertyRef for <code>.name</code>

Elements

1. Context selector (cs:)

Optional prefix naming which stack holds the **receiver**:

Prefix	Stack
<i>(none)</i>	DS by default for reads and ident-rooted stores (<code>sess.config.paramA!</code>); stores onto a receiver parked on DS need <code>t:</code> / <code>x:</code> / ...
<code>t:</code> <code>x:</code> <code>y:</code> <code>z:</code>	auxilliary stacks
<code>o:</code>	Object Stack (Final methods only)

```
List.new^ >ts
t:.len@ $ nl
```

See [Context Selectors](#).

2. Root

How the walk starts:

Form	Role
Leading <code>.name...</code>	receiver already on the selected stack
CapCase / ident then	look up word (usually a Type), use its value as focus; ephemeral Type roots are elided when the path finishes
Leading <code>@()</code> / <code>!()</code> / <code>@{}</code> / <code>!{}</code> / <code>?{}</code>	collection op on the receiver

<code>TokenType.TokenEOF \$</code>	<code>// Type root + class const → value on DS</code>
<code>"dup" find .name@ \$</code>	<code>// instance already on DS + property fetch</code>

3. Member segment: .name

Dot plus a member name. What happens depends on the **following affix** (or whether more path follows).

Follow-on	Property	Method
end of path or mid-path continue	fetch value (implied @)	error — need ^
@	fetch (fused)	n/a
!	store (fused)	n/a
^	n/a	invoke (fused)
~	leave PropertyRef	leave MethodRef

Mid-path `.inner` always means “fetch this property and continue with its value”:

```
dc.dict.len $
```

4. Affixes

Segment	Role
@	fetch via PropertyRef (or fused after <code>.name</code>)
!	store via PropertyRef (or fused after <code>.name</code>)
^	invoke MethodRef (must be terminal)
~	materialize ref after <code>.name</code> (must be terminal in that token)

Fused forms you will see constantly:

```
t:.count@      // . count @
t:.count!      // . count !
t:.reset^      // . reset ^
t:.count~      // . count ~
```

Illegal: `t:.count~@` (`~` cannot be followed inside the same token). Use `t:.count~` then `@`.

5. Collection affixes

Each of these is **one** segment cell:

Segment	DS operands	Result
@()	index:i	value
!()	value then index:i	(store)

Segment	DS operands	Result
@[]	indexes (Mat2N: <code>r</code> then <code>c</code>)	value
![]	value then indexes	(store)
@{}	key	value
!{}	value then key	(store)
?{}	key	exists:b

```

0 t:@()
99 0 t:!( )
"name" t:@{}
"Ada" "name" t:!{}
"name" t:?{}

```

Details and type coverage: [Collection Accessors](#).

Receiver and DS roles

Location	Holds
Context stack (<code>t:</code> ...)	receiver for the path
DS	indexes, keys, values to store, and fetched / method results

Common factory then park:

```

DynamicObject.new^ >ts      // inst on DS → TS
"Ada" t:.name!
t:.name@ $ n!
t:drop

```

Stores (`.name!` , `!( )` , `!{ }`) onto a receiver parked on a stack need `t:` / `x:` / `y:` / `z:` / `o:` . Ident-rooted paths do not (`1 sess.config.paramA!`). `l.` / `pls.` stores are the same. A bare `.name!` on DS is rejected (`d:` does not change that).

Class paths vs instance paths

```
// Class – Type name in the path
List.new^
Math.pi
Int.maxInt

// Instance – receiver on a stack
List.new^ >ts
10 t:.append^
t:.len@ $
t:drop
```

Nested walks and elision

Interior objects fetched mid-path are **not** left on stacks. Only the root receiver (when rooted) and the terminal product (on DS) remain.

```
// TypedObject instance on TS with a List field .tags
Person.new^ >ts
List.new^ >xs
"lang" x:.append^
"math" x:.append^
xs> t:.tags!

t:.tags@ >xs
0 x:@() $ nl      // lang
x:drop
t:drop
```

When a property yields another object you will keep using, park it (`>xs`) rather than relying on a one-shot DS value.

Refs and read-modify-write

```
"dup" find >ts
t:.name~
dup @
drop
"DUP" swap !
t:.name@ $ nl      // DUP
t:drop
```

Standalone `@` / `!` / `^` consume the ref. Fused path terminals elide the ref automatically.

Worked examples

Build and inspect a list

```
List.new^ >ts
10 t:.append^
20 t:.append^
30 t:.append^
t:.len@ $ nl          // 3
1 t:@() $ nl          // 20
40 1 t:!( )
1 t:@() $ nl          // 40
t:drop
```

Map upsert and probe

```
StringMap.new^ >ts
1 "a" t:!{}
2 "b" t:!{}
"a" t:@{} $ nl
"c" t:?{} if
  "c" t:@{} $
else
  "missing" $
then
  nl
t:drop
```

DynamicObject fields and ordinals

```
DynamicObject.new^ >ts
1 t:.x!
2 t:.y!
3 t:.z!
t:.y@ $ nl          // 2
1 t:@() $ nl          // 2 (creation order)
"z" t:?{} $ nl      // true
t:drop
```

Class const plus rooted instance work

Type constants and factories use a CapCase root; instance work parks the handle:

```
Math.pi 2 * $ nl

StringBuilder.new^ >ts
"maxInt=" t:.writeString^
Int.maxInt "%d" fmt t:.writeString^
t:.toString^ $ nl
t:drop
```

Final method with o:

```
: Box#label
  o:.title@
;

Box.new^ >ts
"inbox" t:.title!
t:.label^ $ nl
t:drop
```

Inside `Box#label`, `o:.title@` is an ordinary rooted path on OS.

Segment checklist

Allowed punctuation / affix cells inside a compound token:

```
. @ ! ^ ~ @() !() @{} !{} ?{}
```

Rules of thumb:

- no whitespace inside the token
- `^` and `~` only as the last segment of that token
- member names: letter or `_` start; then letters, digits, `_`, `-`, `$`, `?`
- prefer `t:` / `x:` / ... for any sequence of ops on one object

Related: [Properties](#), [Methods](#), [Collection Accessors](#), [Stacks](#).

Compound Types

Final's compound types hold related values together: ordered sequences (**Lists**), keyed tables (**Maps**), **Sets**, named field objects (**DynamicObject** / **StaticObject** / **TypedObject**), **Vecs** and **Mats**, and columnar grids (**Table**).

The [Quick Tour](#) already introduced what each family is for. This section focuses on the **APIs** you use day to day, with short examples. For the path syntax behind `.append^`, `@()`, `!{}`, and friends, see [Objects](#).

Family	Best for	Access
Lists	ordered sequences, stacks/queues	<code>@()</code> / <code>!()</code> , methods
Maps	dictionaries by key	<code>@{}</code> / <code>!{}</code> / <code>?{}</code>
Sets	list/map hybrid, of unique keys	<code>?{}</code>
DynamicObject	small ad-hoc records, json objects	<code>.field</code> and <code>@{}</code> / <code>@(); { ... }</code> builders; <code>.clone^</code>
StaticObject	fixed-schema rows from a DO prototype	same API as DO for schema fields
TypedObject	reusable record types with a schema (and optional Final methods)	<code>.field</code> and <code>@{}</code> / <code>@()</code>
Vecs	1-dimensional containers of fixed size set at construction	index <code>@[]</code> / <code>!()</code>
Mats	2-dimensional tabular containers of fixed size set at construction	<code>flatIndex @()</code> / <code>!()</code> ; <code>row col @[]</code> / <code>![]</code>
Table	columnar grids with stable multi-key sort	<code>append</code> / <code>sortBy*</code> / <code>foreach</code>

Use `help TypeName` in the TUI for the full member list of any type.

Note: `Mat2f` has additional uses and dedicated `mat::` operations; it is documented along with `VecNf` vectors and OLS under [Linear Algebra](#).

Lists

A List is Final's growable ordered sequence. Items are addressed by zero-based index. Most list types share one API; the specialized types mainly change **what may be stored**, and a few add **sort** and **shuffle**.

Creating a list

```
List.new^           // empty list
16 List.newWithCapacity^ // empty, room reserved for 16 items
5 List.newWithLen^   // length 5, slots filled with zeros for that
type
```

`newWithCapacity` is useful when you already know about how many items you will append. `newWithLen` is useful when you will fill the default zero slots by index with `!` `()`.

Common API

These members are available on **List**, **StringList**, and the numeric list types.

Kind	Name	Effect	Description
class	<code>new</code>	<code>(-- list)</code>	empty list
class	<code>newWithCapacity</code>	<code>(cap:i -- list)</code>	empty list with reserved capacity
class	<code>newWithLen</code>	<code>(len:i -- list)</code>	list of the given length, zero/empty filled
prop	<code>len</code>	<code>(-- len:i)</code>	number of items
method	<code>append</code>	<code>(item:a --)</code>	add at the end
method	<code>insertAt</code>	<code>(item:a at:i --)</code>	insert; 0 = front, len = append
method	<code>hasItem</code>	<code>(item:a -- b)</code>	true if any element equals the item
method	<code>deleteAt</code>	<code>(at:i --)</code>	remove at index; later items shift down
method	<code>clear</code>	<code>(--)</code>	remove all items

Kind	Name	Effect	Description
method	enqueue	(item:a --)	insert at the beginning (queue)
method	dequeue	(-- item:a)	remove from the end (queue)
method	push	(item:a --)	same as <code>append</code> (stack)
method	pop	(-- item:a)	remove from the end (stack)
method	peek	(-- item:a)	end item, leave it in place
method	depth	(-- len:i)	synonym for <code>len</code>
method	sort	(--)	sort the list items (ascending)
method	shuffle	(--)	randomly reorder items

Indexed access

```
cs:@()  // ( index:i -- value )  read
cs:!( ) // ( value index:i -- )  write existing slot
```

The receiver stays on the context stack. Writing with `!( )` requires a valid index in `0 .. len-1`; use `append` / `insertAt` to grow the list.

Iteration

Put the list on DS, then `foreach ... /foreach`. The loop value `i` is each element, in order from index `0` to `len-1`.

Typical pattern

```
List.new^ >ts
10 t:.append^
20 t:.append^
30 t:.append^
0 t:@() $ nl          // 10
1 0 t:!( )           // replace index 0 with 1
t:.len@ $ nl          // 3
t:drop
```

The following pages cover what makes each list family different.

List and StringList

Both types share the [common List API](#). They differ only in what they store, and **StringList** adds sorting.

List

List is the general-purpose sequence. Each cell holds **Any**: numbers, strings, Bool, Null, or other objects (including nested lists and maps). Items do not need to be the same type.

```
List.new^ >ts
"hello" t:.append^
42 t:.append^
true t:.append^
null t:.append^
t:.len@ $ nl    // 4
t:drop
```

Use **List** when the element type varies, or when you are building a temporary list of mixed values (for example before rendering Json).

StringList

StringList stores only **String** values. That keeps storage compact for the common “list of text” case, and rejects non-string appends.

```
StringList.new^ >ts
"alpha" t:.append^
"Beta" t:.append^
"gamma" t:.append^
t:.sort^
t:peek foreach
  i $ sp
/foreach
nl
t:drop
```

sort

```
// ( -- )
.sort^
```

StringList#sort orders by Utf-8 byte values. In plain ASCII that puts uppercase before lowercase ("Beta" before "alpha"). It is **not** locale-aware alphabetic sorting.

List (Any cells) does not provide `sort` .

Numeric Lists

Numeric lists use the [common List API](#), but store a single numeric width in each cell. Values are widened to **Int**, **UInt**, or **Float** on the Data Stack when you read them, and narrowed again when you write.

They exist so large sequences of numbers do not pay the cost of a fully general **List** cell. A **ByteList**, for example, is far more compact than a **List** of small integers, and is the natural home for raw byte buffers used with strings and byte streams.

Types

Type	Stores	On DS
<code>Int8List</code>	signed 8-bit	<code>Int</code>
<code>Int16List</code>	signed 16-bit	<code>Int</code>
<code>Int32List</code>	signed 32-bit	<code>Int</code>
<code>IntList</code>	Final <code>Int</code>	<code>Int</code>
<code>ByteList</code>	unsigned 8-bit (0...255)	<code>Int</code>
<code>UInt16List</code>	unsigned 16-bit	<code>Int</code>
<code>UInt32List</code>	unsigned 32-bit	<code>Int</code>
<code>UIntList</code>	Final <code>UInt</code>	<code>UInt</code>
<code>Float32List</code>	32-bit float	<code>Float</code>
<code>FloatList</code>	Final <code>Float64</code>	<code>Float</code>

ByteList is the unsigned 8-bit list — not a signed “char” width. Use **Int8List** when you need signed 8-bit storage. Unicode code points fit in **Int32List** when you want a dense list of them; day-to-day text work usually streams runes instead.

Example

```
5 FloatList.newWithLen^ >ts
3.14 0 t:!( )
2.71 1 t:!( )
1.41 2 t:!( )
0 t:@() $ nl      // 3.14
t:.sort^
t:drop
```

```

ByteList.new^ >ts
65 t:.append^    // 'A'
66 t:.append^
t:.len@ $ nl
t:drop

```

sort

Every numeric list type supports:

```

// ( -- )
.sort^

```

Sorting uses the natural order for that numeric type.

VecNf / **Mat2f** are fixed-shape dense Float arrays for linear algebra, not Lists — they do not grow. See [Linear Algebra](#).

StringKeyValueList

StringKeyValueList is a specialized list of string key / string value pairs. Unlike **Map** / **StringMap**, the **same key may appear more than once**, and insertion order is kept. That matches HTTP headers and query parameters, where repeated keys are legal.

It does **not** use the ordinary List `append` / `@()` API. Use the members below instead.

Creation

```
StringKeyValueList.new^
16 StringKeyValueList.newWithCapacity^
```

API

Kind	Name	Effect	Description
class	<code>new</code>	<code>(-- list)</code>	empty list
class	<code>newWithCapacity</code>	<code>(cap:i -- list)</code>	empty, reserved capacity
class	<code>newWithLen</code>	<code>(len:i -- list)</code>	fixed length
method	<code>add</code>	<code>(value:s key:s --)</code>	always append a new pair (duplicates allowed)

Keyed accessors

Op	Effect	Description
<code>@{}</code>	<code>(key:s -- value:s)</code>	first match; missing key is an error
<code>!{}</code>	<code>(value:s key:s --)</code>	upsert the first matching key (or append if new)
<code>?{}</code>	<code>(key:s -- b)</code>	true if any entry has that key

Use `add` when you intentionally want a second header with the same name. Use `!{}` when you want to replace the first occurrence.

Example

```
StringKeyValueList.new^ >ts
"text/html" "Content-Type" t:.add^
"1" "X-Count" t:.add^
"2" "X-Count" t:.add^          // second X-Count kept
"application/json" "Content-Type" t:!!{} // replaces first Content-Type

"Content-Type" t:@{} $ nl      // application/json

t:peek foreach
  i >xs
  x:.key@ $ ": " $ x:.value@ $ nl
  x:drop
/foreach
t:drop
```

In `foreach`, `i` is a **StringKeyValue** (same shape as **KeyValue**): read `.key@` and `.value@`.

Maps

A Map is a table of key → value entries. The first write of a key creates the entry; later writes replace the value. Reads of a missing key are an error unless you test first with `?{}` or `hasKey`.

Final provides two map types that share the same programming model.

Common API

Kind	Name	Effect	Description
class	<code>new</code>	<code>(-- m)</code>	empty map
prop	<code>len</code>	<code>(-- len:i)</code>	number of entries
method	<code>clear</code>	<code>(--)</code>	remove all entries
method	<code>hasKey</code>	<code>(key:a -- b)</code>	true if the key exists
method	<code>deleteKey</code>	<code>(key:a --)</code>	remove the entry if present

Keyed accessors

Op	Effect	Description
<code>@{}</code>	<code>(key:a -- value)</code>	get; missing key is an error
<code>!{}</code>	<code>(value key:a --)</code>	create or replace
<code>?{}</code>	<code>(key:a -- b)</code>	existence test (same idea as <code>hasKey</code>)

Iteration

Put the map on DS, then `foreach ... /foreach`. Order is **not** defined. The loop value `i` is a **KeyValue** with `.key@` and `.value@`.

Map

Map accepts keys that are comparable primitives (typically **String** or numbers). Values may be any Final type. Compound objects cannot be used as keys.

Prefer **Map** when you need **numeric** (or other non-string) keys.


```
Map.new^ >ts
"red" 1 t:!{}
"green" 2 t:!{}
1 t:@{} $ nl           // red
1 t:.hasKey^ $ nl      // true
1 t:.deleteKey^
t:.len@ $ nl           // 1
t:drop
```

StringMap

StringMap is the usual choice for string-keyed dictionaries. Only a **String** is accepted as a key; values may still be any type. It inherits the common Map members (`len` , `clear` , and so on).

```
StringMap.new^ >ts
100 "count" t:!{}
"Ada" "name" t:!{}
"name" t:@{} $ nl
"missing" t:?{} $ nl   // false
t:peek foreach
  i >xs
  x:.key@ $ "=" $ x:.value@ $ nl
  x:drop
/foreach
t:drop
```

For small named records where field order matters, or where dotted `.field` access is nicer, see [DynamicObject](#). For a reusable schema, see [TypedObject](#).

Sets

A Set is similar to a Map without values. It holds a single instance of any key entered. Sets, like Maps are **unordered**, so you can query for the existence of a value, but not access the values in ordinal order.

Set operators are provided as follows:

- `set::union` – produces a set that contains the elements of both sets
- `set::intersection` – produces a set with only the items in both sets
- `set::difference` – produces a set of the items of the first set, with those in the second set removed from it
- `set::symmetric difference` – is like an XOR, it returns the unique elements in each set, but not those shared by both sets

Final provides two Set objects:

- `Set`, which can take any comparable type, like numerics and `String`
- `StringSet`, which holds `String` values

A Set can be iterated with `foreach ; i` returns the value.

A Set can be created using the List operators `( ... set)` for a Set and `( ... sset)` for a `StringSet`.

A Set is rendered in JSON as an array; the items are in an undefined order. A Set cannot be produced by a JSON string.

Mats

A Mat (Matrix) is a multidimensional container whose shape is set at construction time. Final currently supports 2-dimensional mats that can hold a variety of strongly typed or Any-typed values. The two dimensions of Final's 2d Mats are its rows and columns. The cells are stored in row-major order, meaning the column cells are adjacent in the underlying array holding the values. A Mat cannot be resized once created.

The factory for all Mats has the following form, using Mat2f as an example:

```
2 3 Mat2f.new^ // creates a mat with 2 rows and 3 columns
```

The value in a cell is accessed either by the row/col matrix accessors `@[] / ![]` or using a flat index, where cell index = row*cols + col using the ordinal accessors `@() / !()`.

Mat is iterable, and returns the items of the Mat in flat ordinal order. There are two convenience iterator accessors, which are basically shortcuts for a range operator:

- `getRowIter` // provides the indexes 0..rows-1
- `getColIter` // provides the indexes 0..cols-1

Typically used like this:

```
myMat l.mat!

l.mat.getRowIter^
foreach
  l.mat.getColIter^
  foreach
    j i l.mat@[] // retrieves item at row j col i
    ...
    newVal j i l.mat![]
  /foreach
/foreach
```

The following Mats are provided. Some serve special purposes, as noted

Mat	Cell Type	Stack Type	Notes
Mat2u8	uint8	Int	1
Mat2u16	uint16	Int	1
Mat2u32	uint32	Int	1
Mat2u	uint	Uint	2
Mat2i1	0/1	Int	3, bit-packed storage, can be used for true/false via Truthy semantics
Mat2i4	0-15	Int	3, stored in a Mat2i8 at present, may be packed in future
Mat2i8	int8, 0-255	Int	3
Mat2i16	int16	Int	
Mat2i32	int32	Int	
Mat2i	Int	Int	
Mat2f32	float32	Float	4
Mat2f	Float	Float	5
Mat2c	Color	Color	3
Mat	Any	Any	

Notes:

1. unsigned value set/returned in an Int
2. unsigned value set/returned in a Uint
3. special use in Blitting operations
4. value set/returned in a Float
5. special use in linear algebra, regression/model testing, and mat:: operators

DynamicObject

DynamicObject (DO) is Final's lightweight named-field object. You create fields as you go; there is no separate schema. It is a frequent choice for Json objects, ad-hoc records, and small configuration-style payloads.

Compared with **StringMap**, a DO is typically leaner and faster when the number of fields stays modest (on the order of tens). It also **preserves field creation order**, and supports both name and ordinal access.

Creation

Empty instance:

```
DynamicObject.new^    // ( -- o:DynamicObject )
```

`newdo` is a short alias for the same factory.

{ ... } literal

Prefer the `{ ... }` builders when you already know the fields. `{` marks stack depth (like `( )`). `}` gathers **value / name** pairs down to that mark into a `DynamicObject`. Use `sm}` instead of `}` to build a **StringMap** the same way.

```
{ 1 "x" 2 "y" 3 "z" } >ts
t:.x@ $ sp t:.y@ $ sp t:.z@ $ nl    // 1 2 3
t:drop

{ "Ada" "name" 36 "age" } >jsonp $ nl
{ "London" "city" "UK" "country" } sm} >ts
"city" t:@{} $ nl
t:drop
```

Stack order is **value then name** (same as `!{}`).

Access

Form	Effect	Description
<code>.name!</code>	<code>(value --)</code>	create or set field <code>name</code>
<code>.name@</code>	<code>(-- value)</code>	read field; error if missing
<code>!{}</code>	<code>(value key:s --)</code>	create or set by string key

Form	Effect	Description
@{}	(key:s -- value)	read by key; error if missing
?{}	(key:s -- b)	true if the field exists
@()	(index:i -- value)	read by creation ordinal
!()	(value index:i --)	write by creation ordinal

A field must be created (with `.name!` or `!{}`) before it can be read with `.name@` or `@{}`.

DynamicObject does not expose Map's `len` / `clear` / `hasKey` methods; use `?{}` for existence, and build a fresh DO when you want a clean slate.

Example

```
{ 1 "x" 2 "y" 3 "z" } >ts
t:.x@ $ sp t:.y@ $ sp t:.z@ $ nl      // 1 2 3
0 t:@() $ nl                        // 1 (first field created)
"x" t:?{ } $ nl                      // true
t:drop
```

When you need to grow fields after creation, start empty and set as you go:

```
newdo >ts
1 t:.x!
2 t:.y!
t:drop
```

Cloning (prototypes)

`.clone^` returns a **shallow** copy with the same fields and values. Use a prototype DO with defaults, then clone for each working instance so mutations stay isolated.

```

{ "" "string" 0 "int" } >xs

x:.clone^ >ys
"y mutate" y:.string!
1 y:.int!

x:.clone^ >zs
"z mutate" z:.string!
2 z:.int!

xs> >jsonp $ nl    // prototype unchanged
ys> >jsonp $ nl
zs> >jsonp $ nl

```

For a fixed field set with lower per-instance overhead, mint a [StaticObject](#) from the same prototype via `.newStatic^`.

Iteration

`foreach` walks fields in **creation order**. The loop value `i` is a **KeyValue**: `.key@` is the field name (String), `.value@` is the stored value.

```

{ "Ada" "name" 36 "age" } >ts
t:peek foreach
  i >xs
  x:.key@ $ ": " $ x:.value@ $ nl
  x:drop
/foreach
t:drop

```

A `DynamicObject` is a one-off instance: its field set is not a reusable type. When you need a named type you can mint repeatedly (optionally with Final methods), use [TypedObject](#).

Optimizations for Large Field Sets

Since `DynamicObject` is the object returned when parsing JSON, it is optimized “under the hood” to support almost unlimited field sizes. This is not a typical use case in normal use of DO as an Object-type, but helps it become an efficient hybrid of an Object and a Map for large machine-generated field sets, such as cache key results that can run into the thousands. If DO has more than 64 fields added, it switches under the hood to use a map-like ordinal lookup, rather than an iterated lookup. This operation does not affect how DO behaves or is used. For an object that follows

purely Map-based semantics, the `Final Map` or `StringMap` types are still recommended.

StaticObject

StaticObject (SO) is a fixed-schema record whose **field names live on a prototype DynamicObject**; each instance holds only a dense value array. Access looks like DynamicObject (named fields, keyed and ordinal items, `foreach`), but the field set cannot grow after construction.

Typical uses: many rows that share one schema — Table rows, SQL-style result shapes, directory-listing style records — when you want DO-like accessors with lower per-instance overhead than cloning full DynamicObjects.

Creation

Build (or obtain) a prototype DO, then call `.newStatic^` on that instance:

```
{ "" "name" 0 "age" } >xs           // prototype: field names + defaults
x:.newStatic^ >ts
"Ada" t:.name!
36 t:.age!
t:.name@ $ sp t:.age@ $ nl         // Ada 36
t:drop
x:drop
```

`.newStatic^` is an **instance method on DynamicObject**:

```
proto.newStatic^  // ( -- so:StaticObject )  cs( proto -- proto )
```

It shallow-copies the prototype's current field values into the new StaticObject. Later stores on the instance do not change the prototype's values; later stores on the prototype do not change existing instances.

There is no `StaticObject.new^` factory — every SO is minted from a DO prototype.

Access

Same API as DynamicObject:

Form	Effect	Description
<code>.name!</code>	<code>(value --)</code>	set an existing schema field

Form	Effect	Description
<code>.name@</code>	<code>(-- value)</code>	read field; error if not on the prototype
<code>!{}</code>	<code>(value key:s --)</code>	set by string key (schema field only)
<code>@{}</code>	<code>(key:s -- value)</code>	read by key
<code>?{}</code>	<code>(key:s -- b)</code>	true if the name is on the prototype
<code>@()</code>	<code>(index:i -- value)</code>	read by prototype field ordinal
<code>!()</code>	<code>(value index:i --)</code>	write by ordinal

Difference from DynamicObject: setting an unknown name panics — the schema is frozen at `newStatic`. Use `?{}` to test membership against the prototype's names.

Example (Table-style rows)

```

: makePrototype
  { "" "name" 0 "age" }
;

makePrototype const: proto

: addRow // ( name:s age:i tbl:Table -- )
  >ys
  proto.newStatic^ >ts
  t:.age! t:.name!
  ts> y:.append^
  y:drop
;

proto Table.newFromDynamicObject^ >z
"Ada" 36 z:peek addRow
"Bob" 24 z:peek addRow
z:drop

```

Iteration

`foreach` walks fields in **prototype registration order**. The loop value `i` is a **KeyValue** (`.key@` / `.value@`), same as `DynamicObject`.

Related

- Prototype / ad-hoc DynamicObjects: [DynamicObject](#) (including `.clone^`)
- Named types with typed fields and Final methods: [TypedObject](#)
- Columnar grids: [Table](#)

TypedObject

TypedObject is how you define your own record types in Final: a fixed set of named fields, each with a declared Type, registered in the type catalog. After registration you create instances with `YourType.new^`, read and write fields like other objects, attach **class constants**, and define **Final methods** with a simple naming convention.

Unlike **DynamicObject**, the schema is declared up front and shared by every instance of the type.

Define and register a type

Typical flow:

1. `TypedObject.newTypeDef^` — empty schema shell (`.name` , `.fields` , `.constants`)
2. Fill `.name` and `.fields` (each field value must be a live catalog **Type**, such as `String` or `List`, not a type-name string)
3. Optional: fill `.constants` with `value .name!` (readonly class consts)
4. `TypedObject.makeType^` — seal and register the type
5. Optional: `: YourType#method ... ; / : YourType##classMethod ... ;` — Final methods
6. `YourType.new^` — create instances

Example schema

The schema for a TypedObject is most conveniently defined using paren operators, as in this example:

```
// you will be building DynamicObjects, with nesting
{
  "Person" "name" // your type name

  // now define your fields
  // you provide the Type object of each field (not the name in quotes)
  {
    String "firstName"
    String "lastName"
    Int    "age"
  } "fields"

  // optional: define class readonly constant values
  {
    111 "defaultAge"
  } "constants"
}
TypedObject.makeType^ // your type is now available to use or add methods
to

Person.defaultAge@ $ nl    // 111
```

Create and fill an instance

```
Person.new^ >ts
"Ada" t:.firstName!
"Lovelace" t:.lastName!
36 t:.age!

t:.firstName@ $ sp t:.age@ $ nl
0 t:@() $ sp          // firstName by declaration ordinal
1 t:@() $ nl
t:drop
```

Field stores are checked against the declared type. Accessors:

Form	Notes
<code>.field@</code> / <code>.field!</code>	Named property
<code>@{}</code> / <code>!{}</code> / <code>?{}</code>	Keyed by field name
<code>@()</code> / <code>!()</code>	Ordinal by field declaration order (faster in hot loops)

`foreach` on a `TypedObject` walks fields in **declaration order**; `i` is a **KeyValue**.

Final methods

After `makeType`, define instance methods with `: TypeName#methodName ... ;` and class methods with `: TypeName##methodName ... ;`. The `;` registers the method on the type; **no dictionary word** remains.

The instance (or, for `##`, the Type) is on the **Object Stack**; use the `o:` prefix. (You do not push or drop OS yourself, the runtime takes care of that.)

```
/*
Print the fieldnames and values for the instance.
( -- ) o( inst:Person )
*/
: Person#print
  ( o:.firstName@ o:.lastName@ o:.age@ ) "%s %s (%d)\n" fmt $
;

/*
Create a new Person with defined default values.
( -- person:Person ) o( type:Person )
*/
: Person##newDefault
  o:.new^ >ts
  "anon" t:.firstName!
  "" t:.lastName!
  Person.defaultAge@ t:.age!
  ts>
;
```

```
Person.newDefault^ >ys
y:.print^
"Bilbo" y:.firstName!
"Baggins" y:.lastName!
y:.print^
y:drop
```

Member names are unique across fields, constants, and instance/class methods on that type.

If you include the recommended prefix `/* */` comments before your method, it gets included in your type catalog, for later listing using `help MyType`.

While a Final method runs, `o:` keeps the receiver in view for nested property and method paths.

Base Types

Some Final Types are designed to be used with an instance of a TypedObject, so that the base object can invoke Final callback methods and event handlers on the TypedObject, called the `delegate`. The documentation for such types will specify the need for your delegate instance. When you define such a TypedObject, you include an additional specifier, `baseType`. The following example uses the final-gui SimGame type as an example:

```
// you will be building DynamicObjects, with nesting
{
  "Person" "name" // your type name
  SimGame "baseType" // this will create a `base` field that will be a
  SimGame
  // same as a regular type: fields and optional constants
```

When you create an instance of your type, it automatically creates an instance of the base type, and sets that into the `base` field that is automatically added to your type. The base type will have a field named `del` (for delegate) that it initializes with your TypeDef object. It uses that to forward callback and event method invocations.

A Base Type will typically require you to implement certain named methods on your type. For example, SimGame requires a `#preInit` and `#postInit` pair, which it calls during its `#init`. A base type may include optional methods that are required if you enable a specific feature of the base type. For example, SimGame has options for keyboard and mouse event handling, and when you enable those features, it requires you to define handler methods for those events. See the documentation for individual base types for the properties and methods of the base type, and the methods, and possibly fields it requires you to implement on your delegate TypedObject type.

Most Base Types will have a template in the examples folder that provide stub implementations of the required and optional events, and a stub TypeObject schema. You can copy this template, rename the file and the type name instances in the file, and you will be ready to add your own custom code.

Table

Table is a named-column row collection. Column names are fixed at construction; you append rows afterward. Sort is **stable**, so successive sorts compose (later keys break ties without scrambling earlier order).

Rows can be any objects that expose the schema's columns — commonly **StaticObject** instances minted from a `DynamicObject` prototype, `DynamicObject` clones, `TypedObject` instances, or other typed values with matching properties / `@()` / `@{}`.

Construction

Class method	Effect	Schema source
<code>newFromDynamicObject</code>	<code>(proto:DynamicObject - - table)</code>	property names of the DO (ordinal access for DO/StaticObject rows)
<code>newFromTypedObject</code>	<code>(t:Type -- table)</code>	TypedObject-derived type with <code>@()</code>
<code>newFromType</code>	<code>(t:Type -- table)</code>	any Type's property names (<code>@()</code> and/or <code>@{}</code>)
<code>newFromColumnNames</code>	<code>(names:StringList - - table)</code>	explicit column name list

Members

Kind	Name	Effect / Type	Description
prop	<code>columnNames</code>	<code>StringList</code>	fixed schema names
prop	<code>rows</code>	<code>List</code>	row list
method	<code>append</code>	<code>(row --)</code>	append a compatible row
method	<code>at</code>	<code>(idx:i -- row)</code>	row by ordinal

Kind	Name	Effect / Type	Description
method	<code>sortByOrd</code>	<code>(idx:i sortDirection:b --)</code>	stable sort by column ordinal
method	<code>sortByName</code>	<code>(name:s sortDirection:b --)</code>	stable sort by column name
class prop	<code>SortAsc</code>	<code>Bool (false)</code>	ascending direction for sort methods
class prop	<code>SortDesc</code>	<code>Bool (true)</code>	descending

Sortable cell types: **Bool**, **Int**, **Uint**, **Float**, **String**, **DateTime**. Compared cells must share a type.

Table is **Iterable** — `foreach` walks rows.

Example (DO proto + StaticObject rows)

```

: makePrototype ( -- proto:DynamicObject )
  { "" "name" 0 "age" }
;

makePrototype const: proto
proto Table.newFromDynamicObject^ const: table

: addRow ( name:s age:i -- )
  proto.newStatic^ >ts
  t:.age! t:.name!
  ts> table.append^
;

"Gopher" 13 addRow
"Alice" 20 addRow
"Bob" 24 addRow
"Alice" 55 addRow

table >ts
0 Table.SortAsc t:.sortByOrd^ // sort by name (column 0)
t:.rows >jsonp $
t:drop

```

`proto.newStatic^` builds a **StaticObject**: field names live on the prototype DO; the instance holds a values array. Same `.name!` / `.age@` API as a `DynamicObject` for those fields.

Successive sorts: sort by age, then by name (or the reverse) — stability preserves relative order for ties.

```
table >ts
1 Table.SortAsc t:.sortByOrd^           // by age
"name" Table.SortAsc t:.sortByName^     // then by name
t:drop
```

Notes

- Append does not deeply validate row shape — supply rows that match the schema accessors the Table was built for.
- Prefer a DO prototype + `newStatic` when you want light row instances with a shared column layout.
- For reusable typed rows with Final methods, mint a `TypedObject` type and use `Table.newFromTypedObject^`.
- `dbIter>table` (from a `DbProvider.query^` iterator) builds the Table from the query's `DynamicObject` prototype and appends `StaticObject` rows. See [SQL Databases](#).

Use `help Table` in the TUI for the full member list.

Compound Builder and Filler Operators

Final provides a concise, convenient set of operators to build or fill many of the most-used collections and objects. The operators work interactively in the interpreter, or can be compiled into code. There are three families of operators, each starts with an opener operator, followed by zero or more elements, then a close operator. Close operators can have variants that control the exact type of the collection from the family that is rendered or filled.

- `(` opens a List or Set builder
- `{` opens an Object or Map builder
- `[` opens a Vec or Matrix *filler* (note: not a builder)

When you enter the opening operator, Final marks the location on DS. The closing operator then gathers up all the items placed on DS after the mark, and puts them into the collection. Easy-peasy!

The operators are best explained by example.

List and Set Builders

There are several List terminators:

- `)` builds a List; the elements can be any Final type
- `s)` builds a StringList; the elements must be of String type
- `i)` builds an IntList; the elements must be of Int type
- `b)` builds a ByteList; the element must be of Int type, 0-255

And two Set terminators:

- `set)` builds a Set; the elements can be any **comparable** type: numerics or String
- `sset)` builds a StringSet; the elements must be of String type

```
// ( -- l:List )
( 1 2 "buckle" )

// ( -- sl:StringList )
( "my" "shoe" s)

// ( -- il:IntList )
( 1 2 3 4 5 i)

// ( -- bl:ByteList )
( 0 1 2 255 b)
```

(The set operators behave identically, but produce the requested set type.)

Object Builders

The Object operators build a new `DynamicObject`, or a new `StaticObject` created from a supplied `DynamicObject` prototype, or a `StringMap`. The items must be in pairs: the first element is the value, the second element is the field name.

```
// ( -- do:DynamicObject )
{
  1 "one"
  2 "two"
}

// Creates a StaticObject minted from proto
// In this example, we use the previous DO as the prototype
// The field names used must be defined in the DO
// ( proto:DynamicObject -- so:StaticObject)
{
  1.0 "one"
  2.0 "two"
so}

// Creates a StringMap
// ( -- sm:StringMap )
{
  1 "one"
  2 "two"
sm}
```

Vec and Matrix Fillers

Unlike the `(` and `{` operators, the `[]` operators fill an instance you supply; they do not create a new instance. The pattern of use is creating the instance with the `new`

factory operator, then using the Paren operators to provide the values. Vec values are provided in 0-based order; Matrix values must be supplied in row-major order, i.e. for a 2D Matrix, the first row supplies col0 to col(cols-1), etc.

NOTES:

- every value based on the length of the Vec/Mat must be supplied, otherwise the closer will fault;
- the values must be compatible with the cell type of the matrix; no numeric coercion is done.

```
// fill a Mat2f Matrix
// ( mat:Mat2f )
2 3 Mat2f.new^
[
  0. 1. 2.
  3. 4. 5.
]

// fill a Vec
// ( vec:Vec )
3 VecNf.new^
[ 1. 2. 3. ]
```

Full Vec/Mat API, operators, and OLS: [Linear Algebra](#).

Nesting Operators

You can nest operators to build complex compound objects. There is no practical limit to the nesting level. The rules are:

- the operators must be in family pairs
- you cannot span inner closers across outer opener boundaries eg. no (1 "two" { 3 "three") }
- the inner collection you build must be a valid type for the outer collection you are building

Word Invocations while building/filling

You are allowed to invoke (interpret) or compile (compiling) a word inside unclosed Paren blocks. This can be useful to factor repeated patterns, perhaps with a computation, into a worker word.

```
: gen3 0 2 range foreach i /foreach ;
( -2 -1 gen3 ) // adds 0 1 2
```

Tips, Cautions, and “Gotchas”

Tips

You can use code inside the operators, and this is often useful. For example, to get the value of the “num” field from an object on xs you can do this:

```
( ... x:.num@ ... )
```

You can do computations inside the operators. For example, to fetch two values and add them, putting the sum on a list slot, you can do this:

```
( ... x:.num@ y:.num@ + ... )
```

Cautions and Gotchas

Final provides comprehensive nesting, semantic, and closing checking on block control structures inside Colon words. But in order to retain the flexibility of interactive as well as compiled operation of Paren operators, Final does almost no static checking of the nesting/closing rules of the Paren operators. Therefore, it is incumbent on the developer or agent to take special care in use of the operators.

In general, Final doesn’t care how or how much you mutate DS during the build, **as long as you do not drop any values that were on DS after the opening operator**. Doing so will have undefined effects, and will likely result in a stack fault. eg.

```
// ( val:a -- ? ) will likely fault
( drop ... )
```

The operators only work on DS items added **after** the opening operator; they do not include any values already on DS. If you want to include eg. DS.tos, you could do this:

```
// ( myVal:a -- myVal:a l:List )
( peek ... )
```

If you wanted the top 2 values, in order you could do this:

```
// ( val0:a val1:a -- val0:a val1:a l:List )  
( over over ... )
```

Any more than that, requires moving eg. to an aux stack.

Strings

The [Quick Tour](#) already introduced `String`: immutable Utf-8 text, ASCII-7 as a proper subset, and the caution that `str::len` is a **byte length**, not a character count. Literals (quoted, raw, and rune ticks) are covered under [Interpreter](#). Comparison of two Strings uses Utf-8 **byte** ordering ([Basic Operators](#)).

This section is the day-to-day **API** for working with text:

Page	Contents
Unicode and Utf-8	How Utf-8 is coded, why finds stay unambiguous, and why that does not slow ordinary string work
String Operators	The full <code>str::</code> family: length, trim, case, concat, probes, split/join
Iterating Runes	<code>foreach</code> over a String (code points as <code>Int</code>) plus a worked census example
StringBuilder	Mutable accumulation when many concatenations would be costly

A few design points that shape every API below:

- **Immutability** — every `str::` “mutation” returns a **new** String; the inputs are unchanged.
- **Stack operators, not instance methods** — text helpers are the `str::...` words (same family style as other axiom groups). Park a String on **ts** only when you need several rooted object ops around it; for pure string transforms, keep values on **DS**.
- **No byte-indexed substring** — cutting at an arbitrary byte offset can split a multi-byte character. Build pieces with `StringBuilder`, use `str::indexOf` / `str::sub` / `str::split`, or iterate runes (this section and [TextReader](#)).
- **Truthy** — `""` is false; any non-empty String is true ([Basic Types](#)).

For line-oriented or file-backed text, prefer [Text and Byte Streams](#). For pattern matching, see [Regex](#) (later in the survey).

Unicode and Utf-8

Final stores every **String** as a sequence of **Utf-8** bytes. A **Rune** (documented under [Basic Types](#) as an **Int** synonym) is a Unicode **code point** — the abstract character value — not a byte. One Rune encodes as **one to four** Utf-8 bytes.

You already know that ASCII-7 is a proper subset of Utf-8: English source, Json keys, and HTML tags that stay in ASCII look identical as bytes whether you call them “ASCII” or “Utf-8.” Final source itself is Utf-8; old DOS/Windows “code pages” are not accepted ([Interpreter](#)).

How the encoding works

Utf-8 uses the **high bits** of each byte to mark where characters start and how long they are:

Byte pattern (binary)	Role	Payload bits
0xxxxxxx	single-byte character (U+0000... U+007F, i.e. ASCII)	7
110xxxxx	start of a 2 -byte sequence	5 + ...
1110xxxx	start of a 3 -byte sequence	4 + ...
11110xxx	start of a 4 -byte sequence	3 + ...
10xxxxxx	continuation byte (never a start)	6

Examples:

- 'A' → one byte 0x41
- 'é' (U+00E9) → two bytes 0xC3 0xA9
- '目' (U+65E5) → three bytes
- '😄' (U+1F642) → four bytes

Only the **shortest** encoding of a code point is legal. Overlong forms (for example encoding ASCII as a two-byte sequence) are invalid. That rule is part of what keeps the encoding unambiguous.

Qualities that matter in practice

These properties are why Final (like Go) can treat Utf-8 as the everyday string encoding without a large runtime tax.

Deterministic find; no ambiguous overlaps

Because ASCII bytes **never** appear inside a multi-byte sequence, a search for an ASCII needle — `"/"`, `".fnl"`, `"\n"`, a JSON delimiter — cannot false-match in the middle of an emoji or accented letter. `str::startsWith`, `str::endsWith`, `str::indexOf`, and `str::split` are byte scans that stay **boundary-safe** when the probe itself is well-formed Utf-8 (including plain ASCII).

The same bit pattern that marks continuation bytes (`10xxxxxx`) also makes Utf-8 **self-synchronizing**: from a random offset you can walk forward to the next start byte without restarting from the beginning of the string. That is the engineering reason “find” and “split on ASCII” stay simple and reliable.

ASCII-majority text stays fast

Most Final programs spend their time on source text, paths, protocol tokens, and similar ASCII-heavy material. For those, length checks, concatenation, prefix/suffix tests, and splits are ordinary **byte** operations — the same cost model as a classic C string, without a hidden per-character decode on every call.

You pay for decoding when you **ask** for characters: `foreach` over a String, `TextReader.getNextRune`, or converting with `r>s` before `$ / $$`. That cost is linear in the text you walk, and for ASCII the decoder’s fast path is essentially “one byte → one rune.”

Kernighan and Donovan’s *The Go Programming Language* walks through this model in detail (strings as bytes, range/decode for runes, and why indexing by “character” is the wrong default). The short version for Final users: **Utf-8 is not a tax on every string op**; it is a compact encoding with a clear, cheap ASCII fast path and an explicit decode step when you need code points.

What “length” means

Question	Answer in Final
How many bytes of storage?	<code>str::len</code>

Question	Answer in Final
How many code points (Runes)?	Count them with <code>foreach</code> (see Iterating Runes)
How many grapheme clusters (user-perceived characters)?	Not a core API — combining marks and emoji sequences need a higher-level library

Because multi-byte Runes exist, `str::len` and the rune count are different whenever the text leaves ASCII. That is expected, not a bug.

Safety model (short)

Operation	Safe for general Unicode?	Notes
<code>str::startsWith</code> / <code>endsWith</code> / <code>indexOf</code> / <code>split</code>	Yes for well-formed Utf-8 probes	Byte match; ASCII probes cannot land inside a multi-byte rune
<code>str::indexOf</code> result	Byte offset	Useful for ASCII wire tokens; do not invent a byte slice from it
Byte-indexed substring	Not provided	Would risk splitting a character; build with StringBuilder or walk runes
Ordering (<code><</code> <code>></code> on Strings)	Byte order	Fine for ASCII; not locale collation

Next: the concrete [String Operators](#).

String Operators

All everyday String helpers are **str:: stack operators**. They take String operands on **DS** and leave results on **DS**. Non-String inputs panic at runtime.

Operands are never modified in place: “mutating” words return a **new** String.

Type `list-string` and then eg `help str::len` (and friends) in the UI for the operator description and stack effects.

Inquiry

Word	Effect	Description
<code>str::len</code>	<code>(s:String -- len:i)</code>	Byte length of the Utf-8 encoding (not rune count)
<code>str::isEmpty</code>	<code>(s:String -- b)</code>	true if <code>s</code> has zero bytes

```
"hello" str::len $ nl      // 5
"é" str::len $ nl         // 2 (one rune, two bytes)
"" str::isEmpty $ nl      // true
```

Empty String is also false under Truthiness, so `str::isEmpty` and a bare `if` on the string agree for control flow — prefer `str::isEmpty` when you want the intent obvious.

Trim

Whitespace follows Unicode space classes (not only ASCII `' '`).

Word	Effect	Description
<code>str::trimStart</code>	<code>(s -- s')</code>	drop leading whitespace
<code>str::trimEnd</code>	<code>(s -- s')</code>	drop trailing whitespace
<code>str::trim</code>	<code>(s -- s')</code>	drop both ends

```
" x.fnl " str::trim ".fnl" str::endsWith $ nl // true
```

Case

Applies Unicode casing rules (not ASCII-only).

Word	Effect	Description
<code>str::toLower</code>	<code>(s -- s')</code>	lowercase
<code>str::toUpper</code>	<code>(s -- s')</code>	uppercase

```
"Hello" str::toLower $ nl    // hello
"αβγ" str::toUpper $ nl     // ΑΒΓ
```

Substitution

Word	Effect	Description
<code>str::sub</code>	<code>(str:s find:s replace:s -- str')</code>	replace every occurrence of <code>find</code> with <code>replace</code>

Returns a **new** String. If `find` is absent, the result equals `str`. Substitution is exact string match (not Regex).

```
"banana" "a" "o" str::sub $ nl    // bonono
"foo" "x" "y" str::sub $ nl       // foo
"aa" "a" "aa" str::sub $ nl      // aaaa
```

For pattern-based replace, see [Regex](#).

Concatenation

Word	Effect	Description
<code>str::cat</code>	<code>(a:String b:String -- ab:String)</code>	concatenate <code>a</code> then <code>b</code>
<code>str::+</code>	<code>(a:String b:String -- ab:String)</code>	synonym for <code>str::cat</code>

```
"hello" " " str::+ "world" str::+ $ nl    // hello world
```

For a few pieces, `str::+` is fine. For building a large result in a loop, use [StringBuilder](#) instead of repeated `str::+` (each concat allocates a fresh String).

Prefix / suffix / find

Word	Effect	Description
<code>str::startsWith</code>	<code>(s:String prefix:String -- b)</code>	true if <code>s</code> begins with <code>prefix</code>
<code>str::endsWith</code>	<code>(s:String suffix:String -- b)</code>	true if <code>s</code> ends with <code>suffix</code>
<code>str::contains</code>	<code>(s:String sub:String -- b)</code>	true if <code>s</code> contains <code>sub</code>
<code>str::indexOf</code>	<code>(s:String sub:String -- idx:i)</code>	byte index of first <code>sub</code> , or <code>-1</code> if absent

```
"internal/final/core/0046-parse.fnl" >ts
t:peek "internal/" str::startsWith $ nl      // true
t:peek ".fnl" str::endsWith $ nl             // true
t:peek "/" str::indexOf $ nl                  // 8
t:drop

"banana" "an" str::contains $ nl             // true
"banana" "xy" str::contains $ nl             // false
```

Split and join

Word	Effect	Description
<code>str::split</code>	<code>(s:String sep:String -- parts:StringList)</code>	split on exact separator sequence
<code>str::splitLines</code>	<code>(s:String -- lines:StringList)</code>	split on <code>\n</code> ; one trailing <code>\n</code> is stripped before splitting
<code>str::join</code>	<code>(items:List StringList sep:String -- s:String)</code>	join string items with <code>sep</code>

`str::split` and `str::join` round-trip cleanly with **StringList**. `str::join` also accepts a **List** whose every element is a **String** (non-String items panic).

```
"a,b,c" "," str::split >ts
t:peek foreach
  i $ sp
/foreach
nl
"," t:peek str::join $ nl    // a,b,c
t:drop
```

```
"alpha\nbeta\n" str::splitLines >ts
t:.len@ $ nl          // 2
0 t:@() $ nl          // alpha
1 t:@() $ nl          // beta
t:drop
```

Notes on `str::splitLines`:

- The empty String yields an **empty** StringList (not a single empty line).
- A final `\n` does not create an extra empty trailing line.
- Only `\n` is the separator; a lone `\r` is left in the line text (CRLF files are usually normalized by [TextReader.readLine](#), which strips a trailing `\r`).

Patterns you will see in core

Error messages and path checks often chain concat and probes without parking:

```
"WARNING: word not found: " "compileFromFile" str::+ $ nl

"notes.fnl" ".fnl" str::endsWith not if
  "expected a .fnl path" $ nl
then
```

When the next step is list processing (sort, foreach, json), keep the **StringList** from `str::split` / `str::splitLines` on **ts** and use the [List API](#).

Next: [Iterating Runes](#) when you need code points rather than bytes.

Iterating Runes

A **Rune** is a Unicode code point carried as an **Int** ([Basic Types](#)). To walk the characters of a String — not its raw bytes — put the String on **DS** and use **foreach** ... **/foreach**. The loop variable **i** is each successive Rune.

You already met **foreach** for lists and maps in [Iterators: foreach](#). **String** participates in the same protocol: **foreach** obtains a fresh rune iterator and drives the loop until the text is exhausted.

```
"aé😊☺" foreach
  i r>s $ sp
/foreach
nl
// prints: a é 😊 ☺
```

Print a single Rune with **r>s \$ (r>s is (rune:i -- s) ; \$** prints the String). Ordinary **\$** on an **Int** prints the decimal code point, which is rarely what you want for text. Inside a **fmt** string, use **%c** with the Int code point instead of converting first.

TextReader is also iterable by rune ([Stream Types](#)). Prefer **foreach** on a **String** when the text is already in memory; prefer a reader when the source is a file or you want line APIs as well.

Byte length vs rune count

```
: runeCount ( s:String -- n:i )
  0 swap
  foreach
    drop
    1 +
  /foreach
;

"é" dup str::len $ sp    // 2
runeCount $ nl          // 1
```

That gap is exactly the Utf-8 story from [Unicode and Utf-8](#): **str::len** is storage; **foreach** is characters.

Example: source file rune census

This small program reads a Utf-8 source file into a String, walks every Rune, and prints a compact summary: how many code points, how the Utf-8 widths break down, and which non-ASCII characters appear. It is a handy sanity check when a file “looks fine” in an editor but something downstream complains about encoding — and it shows `foreach`, `str::len`, and **StringBuilder** working together.

```

: utf8Width ( r:i -- n:i )
  dup 0x80 < if
    drop 1
  else
    dup 0x800 < if
      drop 2
    else
      0x10000 < if
        3
      else
        4
      then
    then
  then
;

: readFileString ( path:s -- s:String true | err:s false )
  TextReader.newFromFile?^ not if
    false
  else
    >ts
    StringBuilder.new^ >xs
    t:peek foreach
      i x:.writeRune^
    /foreach
    t:.close^
    t:drop
    x:.toString^
    x:drop
    true
  then
;

: printRuneCensus ( path:s -- )
  dup >ys
  readFileString not if
    "cannot read: " y:peek str::+ $ nl
    " " swap str::+ $ nl
    y:drop
  else
    >xs
    newdo >ts
      0 t:.runes!
      0 t:.w1!
      0 t:.w2!
      0 t:.w3!
      0 t:.w4!
    StringBuilder.new^ >zs
    x:peek foreach
      t:.runes@ 1 + t:.runes!

```

```

i utf8Width
switch
  a 1 == case
    t:.w1@ 1 + t:.w1!
  /case
  a 2 == case
    t:.w2@ 1 + t:.w2!
  /case
  a 3 == case
    t:.w3@ 1 + t:.w3!
  /case
  default
    t:.w4@ 1 + t:.w4!
  /default
/switch
i 0x80 >= if
  i z:.writeRune^
  32 z:.writeRune^
then
/foreach
"File: " y:peek str::+ $ nl
"  bytes (str::len): " x:peek str::len "%d" fmt str::+ $ nl
"  runes:           " t:.runes@ "%d" fmt str::+ $ nl
"  utf-8 widths:    " $
  t:.w1@ "%d×1 " fmt $
  t:.w2@ "%d×2 " fmt $
  t:.w3@ "%d×3 " fmt $
  t:.w4@ "%d×4" fmt $ nl
"  non-ASCII runes: " $
z:.toString^ dup str::isEmpty if
  drop "(none)" $ nl
else
  $ nl
then
z:drop
t:drop
x:drop
y:drop
then
;

"examples/stringRune-iter.fn1" printRuneCensus

```

Sample shape of the report (counts depend on the file):

```
File: examples/stringRune-iter.fnl
bytes (str::len): 218
runes:           210
utf-8 widths:    204×1 2×2 0×3 4×4
non-ASCII runes: é 😊 ☐
```

The interesting lines are the ones where **bytes ≠ runes** and the width histogram is not “all `×1`”: that is Utf-8 doing its job, and `foreach` making the character stream easy to measure without hand-rolled decoding.

Next: [StringBuilder](#) for the mutable side of building text.

StringBuilder

StringBuilder is the mutable partner to immutable **String**. Append runes, strings, or lines as you go, then take a snapshot with `toString`. Use it when a loop would otherwise call `str::+` over and over (each concat allocates a new String).

You already saw a short builder example under [Methods](#) and the stream-shaped write members under [String Streams](#). This page is the full API.

Park the builder on **ts** (or another aux stack) for a session of writes, then `t:drop` — same nesting style as other objects ([Writing Clean Final Code](#)).

Factories

Kind	Name	Effect	Description
class	<code>new</code>	<code>(-- sb)</code>	empty builder
class	<code>newWithPretty</code>	<code>(indent:String -- sb)</code>	pretty mode: indentation string for nested structured output

```
StringBuilder.new^
" " StringBuilder.newWithPretty^
```

Pretty builders are aimed at structured renderers (for example Json). For ordinary path joins, messages, and file assembly, plain `new` is enough.

Properties

Name	Type	Description
<code>pretty</code>	Bool	<code>true</code> if created with <code>newWithPretty</code> (read-only)
<code>indentString</code>	String	indent unit used when pretty (read-only; empty for a plain builder)

Write and snapshot

Kind	Name	Effect	Description
method	<code>writeRune</code>	<code>(r:i --)</code>	append one Unicode code point

Kind	Name	Effect	Description
method	<code>writeString</code>	<code>(s:String -)</code>	append as-is (no added newline)
method	<code>writeLine</code>	<code>(s:String -)</code>	append string, then <code>\n</code>
method	<code>toString</code>	<code>(-- s:String)</code>	current contents as an immutable String

```

StringBuilder.new^ >ts
"hello" t:.writeString^
32 t:.writeRune^
"world" t:.writeLine^
t:.toString^ $           // hello world\n
t:drop

```

`toString` does not clear the builder; further writes keep appending. Build a fresh **StringBuilder** when you want a clean buffer.

Pretty / nesting helpers

Only meaningful when **pretty** is true (`newWithPretty`). Plain builders treat the pretty writes as ordinary writes (or no-ops where noted).

Kind	Name	Effect	Description
method	<code>writeStringPretty</code>	<code>(s:String -)</code>	write current indent (if nested), then <code>s</code>
method	<code>writeSpacePretty</code>	<code>(--)</code>	write a space if pretty; otherwise nothing
method	<code>writeWhitespacePretty</code>	<code>(--)</code>	write <code>\n</code> if pretty, else a single space
method	<code>nest</code>	<code>(--)</code>	increase indent level by one
method	<code>unnest</code>	<code>(--)</code>	decrease indent level by one (stops at 0)

```

" " StringBuilder.newBuilderWithPretty^ >ts
{" t:.writeString^
t:.writeWhitespacePretty^
t:.nest^
\"name\" t:.writeStringPretty^
t:.writeSpacePretty^
": t:.writeString^
t:.writeSpacePretty^
\"Ada\" t:.writeString^
t:.unnest^
t:.writeWhitespacePretty^
"} t:.writeString^
t:.toString^ $ nl
t:drop

```

Typical pretty output:

```

{
  "name" : "Ada"
}

```

When to choose what

Need	Prefer
One or two concatenations	<code>str::+</code> / <code>str::cat</code> (String Operators)
Many appends in a loop	StringBuilder
Character-at-a-time parse/build	Runes via <code>foreach</code> / TextReader , append with <code>writeRune</code>
Line-oriented file I/O	TextReader / TextWriter (Streams)
Indented structured text	<code>newWithPretty</code> + <code>nest/unnest</code>

Strings remain Utf-8 end to end: every `writeRune` encodes a code point; `toString` yields a normal immutable **String** ready for `str::` operators or `$`.

Using a StringBuilder as a TextWrite value

StringBuilder implements the TextWrite interface. This means you can supply a StringBuilder to contexts that accept a TextWrite, such as when you want to capture the stdout/stderr of a [Pip \(thread\)](#); or in Http bodies expecting a TextWrite Stream.

DateTime and Interval

The [Quick Tour](#) deferred these two basic types here. **DateTime** is a calendar date and/or clock time; **Interval** is a length of time (a duration). Both are **immutable** sealed values.

They are **not** part of the numeric family. Ordinary `+` `-` `*` `/` and compares do **not** apply — use the `dt::` family below. Invalid operand pairs panic.

Related material already covered elsewhere:

- `$` can print a DateTime ([stdin/out Streams](#)).
- Json rendering emits DateTime as an RFC 3339 string ([Json Rendering](#)).
- Pattern formatting for DateTime and Interval is covered in [Formatting with dt::fmt](#) below; general `%` message formatting is in [Output Formatting](#).

Type `list-datetime` for all the DateTime/Interval related words, then e.g. `help DateTime`, `help Interval`, `help dt::+`, or `help dt::fmt` in the UI for the detailed documentation and stack effects.

DateTime

Factories create values in **UTC**. `DateTime.now` returns **UTC** as well (the current instant).

Construction takes a **List of Int components** — typically written with `( ... )` — rather than a long stack of loose numbers.

Kind	Name	Effect	Description
class	<code>now</code>	<code>(-- dt:DateTime)</code>	current instant (UTC)
class	<code>newDate</code>	<code>(parts:List -- dt:DateTime)</code>	year, month (1-12), day → midnight UTC
class	<code>newTime</code>	<code>(parts:List -- dt:DateTime)</code>	hour, minute, second, optional ms → clock-only
class	<code>newDateTime</code>	<code>(parts:List -- dt:DateTime)</code>	year...second, optional ms → full instant UTC

Kind	Name	Effect	Description
prop	<code>year</code>	Int	calendar year
prop	<code>month</code>	Int	month 1–12
prop	<code>day</code>	Int	day of month
prop	<code>hour</code>	Int	hour 0–23
prop	<code>minute</code>	Int	minute 0–59
prop	<code>second</code>	Int	second 0–59
prop	<code>nanosecond</code>	Int	nanoseconds within the second
prop	<code>isZero</code>	Bool	<code>true</code> if the zero <code>DateTime</code>
method	<code>since</code>	<code>(-- iv:Interval)</code>	elapsed Interval from this instant until now

```
( 2026 7 4 ) DateTime.newDate^ $ n\
( 14 30 0 ) DateTime.newTime^ $ n\
( 2026 7 4 14 30 0 250 ) DateTime.newDateTime^ $ n\
DateTime.now^ $ n\
```

Park a `DateTime` on **ts** when you need several property reads:

```
DateTime.now^ >ts
t:.year@ $ sp t:.month@ $ sp t:.day@ $ n\
t:drop
```

Interval

An Interval is a **duration**. Build one from unit class constants (then scale with `dt::*`), or from a List of clock-length components.

Kind	Name	Effect	Description
class	<code>nanosecond</code>	<code>(-- iv:Interval)</code>	one nanosecond
class	<code>microsecond</code>	<code>(-- iv:Interval)</code>	one microsecond

Kind	Name	Effect	Description
class	<code>millisecond</code>	<code>(-- iv:Interval)</code>	one millisecond
class	<code>second</code>	<code>(-- iv:Interval)</code>	one second
class	<code>minute</code>	<code>(-- iv:Interval)</code>	one minute
class	<code>hour</code>	<code>(-- iv:Interval)</code>	one hour
class	<code>newInterval</code>	<code>(parts:List -- iv:Interval)</code>	hour, minute, second, optional ms
prop	<code>nanoseconds</code>	<code>Int</code>	total nanoseconds
prop	<code>microseconds</code>	<code>Int</code>	total microseconds (truncated)
prop	<code>milliseconds</code>	<code>Int</code>	total milliseconds (truncated)
prop	<code>seconds</code>	<code>Float</code>	total seconds, including fraction
prop	<code>minutes</code>	<code>Float</code>	total minutes, including fraction
prop	<code>hours</code>	<code>Float</code>	total hours, including fraction
prop	<code>days</code>	<code>Float</code>	total days, including fraction (<code>hours / 24</code>)

```
Interval.second $ nl
5 Interval.second dt::* $ nl           // 5s
5 Interval.second dt::* .milliseconds@ $ nl // 5000
( 1 30 0 ) Interval.newInterval^ $ nl // 1h30m0s
( 0 0 5 500 ) Interval.newInterval^ $ nl
```

Unit constants are class properties (same style as `Math.pi` / `Int.maxInt` — see [Properties](#)).

dt:: operators

All take operands on **DS**. Same-kind compares only (two DateTimes, or two Intervals). Mixed kinds and unsupported arithmetic pairs **panic**.

Word	Effect	Valid pairs / notes
<code>dt::+</code>	<code>(a b -- result)</code>	<code>DateTime + Interval → DateTime</code> (either order); <code>Interval + Interval → Interval</code>
<code>dt::-</code>	<code>(a b -- result)</code>	<code>DateTime - Interval → DateTime</code> ; <code>DateTime - DateTime → Interval</code> ; <code>Interval - Interval → Interval</code>
<code>dt::*</code>	<code>(a b -- iv:Interval)</code>	<code>Int × Interval</code> (either order)
<code>dt::/</code>	<code>(iv:Interval n:i -- iv:Interval)</code>	divide by positive Int (truncates toward zero); panics if <code>n <= 0</code>
<code>dt::compare</code>	<code>(a b -- c:i)</code>	-1 / 0 / 1 (before / equal / after)
<code>dt::<</code> <code>dt::<=</code> <code>dt::></code> <code>dt::>=</code> <code>dt::==</code> <code>dt::</code> <code><></code>	<code>(a b -- c:b)</code>	ordering / equality; both either <code>DateTime</code> or <code>Interval</code>
<code>dt::round</code>	<code>(iv:Interval roundTo:Interval -- rounded:Interval)</code>	nearest multiple of <code>roundTo</code>
<code>dt::fmt</code>	<code>(val:DateTime Interval fmt:s -- result:s)</code>	pattern format (see below)

Formatting with dt::fmt

`$` prints a `DateTime` or `Interval`'s **natural** string. For a composed layout — ISO, a prose date, a stopwatch clock — use `dt::fmt` with a **pattern String**.

```
dt::fmt // ( val:DateTime|Interval fmt:s -- result:s )
```

- One word for both types; the lexicon is chosen from **val's type**.
- `fmt` is always a pattern String (or a class constant that holds one). Wrong tokens for that type panic.
- Patterns use letter-run codes familiar from app languages (`yyyy-MM-dd` , `HH:mm:ss`).

```
DateTime.now^ "yyyy-MM-ddTHH:mm:ss.sssZ" dt::fmt $ nl
DateTime.now^ DateTime.iso dt::fmt $ nl
5 Interval.minute dt::* "HH:mm:ss" dt::fmt $ nl
Interval.second Interval.clock dt::fmt $ nl // 00:00:01
```

General message formatting (`fmt / %d / %s`) is separate — see [Output Formatting](#).

DateTime patterns

Case-sensitive. `MM` $\neq$ `mm`; `HH` $\neq$ `hh`; `a` $\neq$ `A`. Values are formatted in **UTC**.

Token / fragment	Meaning	Example
<code>yyyy</code>	4-digit year	<code>2026</code>
<code>yy</code>	2-digit year	<code>26</code>
<code>M</code>	month 1–12 (unpadded)	<code>7</code>
<code>MM</code>	month 01–12	<code>07</code>
<code>MMM</code>	short month name (English)	<code>Jul</code>
<code>MMMM</code>	long month name (English)	<code>July</code>
<code>d</code>	day of month (unpadded)	<code>4</code>
<code>dd</code>	day of month 01–31	<code>04</code>
<code>EEE</code>	short weekday (English)	<code>Sat</code>
<code>EEEE</code>	long weekday (English)	<code>Saturday</code>
<code>HH</code>	hour 00–23 (24-hour)	<code>14</code>
<code>hh</code>	hour 01–12 (12-hour)	<code>02</code>
<code>mm</code>	minute	<code>30</code>
<code>ss</code>	whole seconds	<code>05</code>

Token / fragment	Meaning	Example
<code>ss.sss ...</code>	seconds + fraction (<code>s</code> run = digit width; ms / μ s / ns)	<code>05.250</code>
<code>a</code>	meridiem lower	<code>am / pm</code>
<code>A</code>	meridiem upper	<code>AM / PM</code>

Hours / meridiem: `hh` does **not** add AM/PM by itself — include `a` or `A` when you want it.

Fractional seconds: after `ss`, an optional `.` plus a run of `s` sets the digit width (`ss.sss` = milliseconds, up to nine `s` for nanoseconds). A `.` elsewhere in the pattern is a literal.

Literals (no quotes needed): space, `T`, `Z`, and usual separators (`-`, `:`, `/`, `,`). `z` is the letter Z (UTC designator in ISO-style patterns), not a computed offset.

Quoted literals: wrap text in `'...'` when you need a letter or `.` that would otherwise be a token (e.g. `ss'.'` for a literal dot after seconds with no fraction). `'` inside quotes is a single `'`.

```
DateTime.now^ "EEEE, MMMM d, yyyy" dt::fmt $ n\l
DateTime.now^ "hh:mm A" dt::fmt $ n\l
DateTime.now^ "yyyy-MM-ddTHH:mm:ss.sssZ" dt::fmt $ n\l
```

DateTime format constants

Class properties (same style as `Math.pi`) — each pushes a pattern String:

Constant	Pattern
<code>DateTime.iso</code>	<code>yyyy-MM-ddTHH:mm:ss.sssZ</code>
<code>DateTime.date</code>	<code>yyyy-MM-dd</code>
<code>DateTime.time12</code>	<code>hh:mm:ss A</code>
<code>DateTime.time24</code>	<code>HH:mm:ss</code>
<code>DateTime.dateTime</code>	<code>yyyy-MM-dd HH:mm:ss</code>
<code>DateTime.longDate</code>	<code>EEEE, MMMM d, yyyy</code>
<code>DateTime.shortDate</code>	<code>MMM d, yyyy</code>

```
DateTime.now^ DateTime.iso dt::fmt $ n\l
DateTime.now^ DateTime.longDate dt::fmt $ n\l
```

Interval patterns

Case-insensitive (the runtime lowercases the pattern before parsing). You may write `HH` or `hh` — same meaning. No meridiem, no calendar month/year names, no weekday tokens.

Token (after lower)	Meaning	Padding
<code>d</code> / <code>dd</code> / ...	days (24h chunks only — not months/years)	width = run length; space -padded on the left (<code>dd</code> + 5 → <code>" 5"</code> ; <code>d</code> → <code>"5"</code>)
<code>h</code> / <code>hh</code> / ...	hours	min width 2, zero -padded; widens if the value needs more digits
<code>m</code> / <code>mm</code> / ...	minutes	same zero-pad / widen rule
<code>s</code> / <code>ss</code> / ...	seconds	same; optional <code>. + s ...</code> fraction (same idea as <code>DateTime</code>)
<code>+</code>	sign placeholder	≥ 0 → space; < 0 → <code>-</code> (column-aligned)
space, <code>:</code> , other non-letters	literals	—

Cascade: only units that **appear in the pattern** participate. Among those, the **largest** unit absorbs overflow; smaller units are remainders. Absent units are not filled in silently.

Duration	Pattern	Output
50h	<code>hh:mm</code>	<code>50:00</code>
50h	<code>d hh:mm</code>	<code>2 02:00</code>
50h	<code>dd hh:mm</code>	<code>2 02:00</code>
50h	<code>mm</code>	<code>3000</code>

Sign:

- Format the **absolute** duration with the numeric fields.
- If the pattern contains `+`, emit space when `val ≥ 0`, `-` when `val < 0`, at that position.
- If there is **no** `+` and the value is negative, prefix a single `-`.
- Zero is non-negative → space when `+` is present.

```
( 1 30 0 ) Interval.newInterval^ "hh:mm:ss" dt::fmt $ nl      // 01:30:00
50 Interval.hour dt::* "d hh:mm" dt::fmt $ nl                // 2 02:00
50 Interval.hour dt::* "dd hh:mm" dt::fmt $ nl                // 2 02:00
1250 Interval.millisecond dt::* "ss.sss" dt::fmt $ nl         // 01.250

// negative: leading - when no + in pattern
5 Interval.hour dt::* -1 dt::* "d hh:mm:ss" dt::fmt $ nl     // -0
05:00:00
5 Interval.hour dt::* -1 dt::* "+d hh:mm:ss" dt::fmt $ nl     // -0
05:00:00
```

Interval format constants

Constant	Pattern	Role
<code>Interval.clock</code>	<code>hh:mm:ss</code>	stopwatch-style
<code>Interval.clockMs</code>	<code>hh:mm:ss.sss</code>	with milliseconds
<code>Interval.daysClock</code>	<code>d hh:mm:ss</code>	day + clock
<code>Interval.signedDaysClock</code>	<code>+d hh:mm:ss</code>	aligned sign

```
( 1 30 0 ) Interval.newInterval^ Interval.clock dt::fmt $ nl
50 Interval.hour dt::* Interval.daysClock dt::fmt $ nl
```

Token overlap

The same letters appear in both lexicons with **different** meanings (`hh` is 12-hour on a `DateTime`, hours-of-duration on an `Interval`). `val`'s **type** selects which table applies. Do not pass a `DateTime` ISO pattern to an `Interval` (or the reverse).

Examples

Elapsed time

Capture a start instant, do work, then subtract with `dt::-`. From a parked start, `.since^` is the same idea (elapsed until now):

```
DateTime.now^ >ts
// ... work ...
DateTime.now^ ts> dt::- $ nl    // elapsed Interval
```

Deadline from now

Advance the current instant by a duration, then compare:

```
DateTime.now^
5 Interval.minute dt::* dt::+ >ts    // deadline = now + 5m

DateTime.now^ t:peek dt::< $ nl      // true while still before deadline
t:drop
```

Average of several Intervals

Scale and divide with `dt::*` / `dt::/` (metrics-style averaging):

```
( 0 0 10 ) Interval.newInterval^    // 10s
( 0 0 20 ) Interval.newInterval^    // 20s
dt::+
2 dt::/ $ nl                        // 15s
```

Notes

- **Immutability** — `dt::+` / `dt::-` and friends return **new** values; inputs are unchanged.
- **No `DateTime + DateTime`** — elapsed time is `dt::-` (later minus earlier), not addition.
- **Printing** — `$` uses the `DateTime` natural string (ISO-style for full dates; clock-only for `newTime` values). `Interval` prints in a compact duration form (e.g. `1h30m0s`). For controlled layouts, use `dt::fmt` and the pattern tables above.
- **Json** — only `DateTime` participates in `>json` / `json>` as a string; see [Json](#).

Word- and Pip-local variables

Final provides the ability to define named variables within a `word`, called `local` or `l variables`, and in the context of a single `Pip (thread)`, called `pip-local` or `pls variables`. The two have different use cases, but operate almost identically, so we cover both here.

Word-local variables via `l`

Word-local variables are helpful in cases where a certain value, sometimes a flag, wants to be available to all words at all block nesting depths within a word. Maintaining such a control variable will typically require dedicating a stack to it, and modifying it requires pops/drops/repushes. They are also useful generally, as a way to provide named access to local values, rather than trying put them on auxiliary stacks, and then try to remember which stack holds which value. A more convenient alternative is a local variable, accessed via the keyword `l`.

`l` accesses a `DynamicObject` that is just-in-time instantiated when you first access `l`. The usual pattern is to initialize your variable or variables on entry to the word, and then query or mutate as needed within the word. The compiler provides special support for `l` by enabling you to just type eg. `false l.myFlag!` then `l.myFlag@` to access it anywhere in the word.

Pip-local storage via `pls`

Pip-local storage works almost identically to Word-local variables, except the scope of visibility and lifetime are that of the Pip, not a word. For the root Pip, they thus function much like a global variable such as `var:`. This feature should be used sparingly.

Since the visibility of `pls` is the entire Pip, and any words that run in it, you should take care to use a top-level field unlikely to be used by or conflict with other words. Consider adding a “namespace” prefix to your field names.

You access Pip-local storage in a similar way as locals, by using the `pls.` keyword.

Here is an example of moving the Pip Params object onto `pls` for availability to all words in your Pip:

```
/*  
Launch word of the Pip.  
Move pp to pls for Pip-global access.  
( pp:PipParams -- )  
*/  
: myPip  
  pls.pp!  
  // remaining init logic  
;
```

Events and the EventLoop

Under the hood, both `final` and `final-gui` use what is called an Event-driven paradigm. When you enter text in the prompt, or load a file, project, or module, the operation is dispatched via an Event sent to Final's core EventLoop.

An event-driven architecture provides benefits that a conventional procedure-driven architecture cannot, and can greatly simplify designs for such as simulations and games, low-level Pi 5 controller programs, and Graphical User Interfaces. All those topics are covered in later chapters, but it is a good idea to get a grounding in this foundation now.

Events

An event is an occurrence such as a key press, mouse movement, regular ticker notification, or single-shot timer firing. Event-driven programs operate quite differently from procedure-driven programs. Look at this startup code for Shape Zombies:

```
: shape-zombies
  CircZomb.new^ >zs
  z:.base >ys // the SimGame base type
  y:.init^
  y:.makeUi^
  y:.startPaused^
  y:.doRender^
  y:.win@.show^
  ydrop zdrop
;
```

Notice that there is no call to a main processing routine—the code shows the main window of the game, then exits. The reason is because the actual program is distributed across a variety of *event handlers*, which are methods that are called by the system when their corresponding event occurs. Here is the `handleKeyDown` event; this gets called when you press a key (a separate event happens when the key is released)

```

/*
KeyDown event handler.
Attached to Window.KeyDown events
( ke:KeyEvent -- ) o( inst )
*/
: CircZomb#handleKeyDown
  CircZomb_onKeyDown
;

```

You can see that the handler routine is passed something called a KeyEvent, which has the name of the key that was pressed. The handler code in onKeyDown then looks for the arrow keys, and if so, moves the position of the user hexagon.

EventLoop

The Final EventLoop is written in Final; it listens on a Pipe for a new event, and when one arrives, dispatches it to the registered handler.

Note that if you block the main thread by executing a continuous loop, or call delayMs with lengthy delays, event processing will stall. For this reason, time-based occurrences are best handled with Timer Events (covered next).

The EventLoop is the most fundamental piece of code in `final / final-gui` – if you do a stack dump with `*$` you can see it sitting on the bottom of the O Stack, since it persists for the entire life of the program.

Events can originate from many sources, and do not have to come from the main thread. UI Events come from its UI thread, which is the application main thread. MotionHit events in SimGame originate on the MotionHitEngine background thread. Timer and Ticker events come from a dedicated thread. But they all get dispatched in the main Pip/thread of your program. (Only the main thread has an EventLoop; Pips cannot use event handlers, but they can send events.)

The Final UI itself is event-based, and uses an event to handle prompts sent from the main window.

Event-based Programming

Note that you are not required to write your Final programs using an event-driven approach, unless you want to use features such as Ticker/Timer events, the Fyne UI, the SimGame framework, or Pi controllers.

Event-base programming is a different paradigm than a single procedural program, but it doesn't take much to get used to it. It is highly beneficial in many ways, because rather than one huge omnibus program with giant switch statements and interminable numbers of sub-words, each event handler in eg the SimGame framework is dedicated to handling just one specific kind of occurrence. The code there can remain focused and simple. Look at the event. Determine what to do (if anything). Do it. End.

Examples

EventTicker and EventTimer Events

see: `examples/events/ticker-timer-demo.fnl`

This demo will run on both versions of Final.

It shows how to:

- create a TypedObject to handle events
- write the handleTicker and Timer events
- create the Ticker and Timer, attaching to the handlers
- start the timers
- handle the timer events

SimGame Events

This is an advanced look at handling events in the [SimGame Framework](#). This program only runs in `final-gui`.

`examples/gui/simgame/invaders.fnl`

Have a look at the various events in Invaders to get a feel for the paradigm. Most event handlers are named `handle<EventName>` such as `handleRender` for drawing, or `handleUpdate` for the periodic update event you can enable, or `handleMouseMove` for mouse movements.

Error and Structured Exception Handling (SEH)

Sometimes, processing cannot continue because some form of error has arisen. Sometimes these are developer-caused, such as integral divide by zero, or passing an incompatible type to an operator or method. Other times it can relate to unexpected service failures, such as a database or web call timeout. And other times it is caused by external input, not under the developer's control, that is nevertheless needed to continue. An example is missing or invalid input data, such as an invalid JSON string.

Final has three (well... two and a half...) ways in which it can deal with errors:

- `panic`, which is used by axioms or other runtime code
- in Final, an `error` call, which uses `panic` under the hood
- an exception, via the `throw` word

Of the three, `panic` (and thus `error`) is the nuclear option. When a `panic` occurs, the runtime does the following:

- if it occurs in a child Pip, it will virally cause all child Pips of the Session to end
- if in the main Session Pip, or propagated up from a child Pip, the main Pip is reset via a "Warm Boot". The stacks are cleared, and the interpreter is then invoked and entered as if from a new Session (the Dictionary remains intact however.)

Final's philosophy is to `panic` or `error` in any situation that could have been prevented; basically all situations of programmer error, such as requesting integral division by zero, or passing a number to a routine that required a String.

In situations involving non-preventable errors, particular those arising from external inputs, Final uses the `throw` statement, which is an error that you can trap by way of a `try / catch` block in your code. `throw` is particularly useful in deeply nested code, in which trying to carry some kind of termination flag, along with the extra code at every level also testing for this flag, results in overly complex code that intermixes normal processing with tests for the error state. A good example is JSON parsing, which can be deeply recursively nested as it processes arrays and objects within outer arrays or objects, or which can encounter an unexpected end of stream before parsing is complete. Final's `json>` operator uses the `throw` statement in many places, when it encounters invalid input, or unexpected end of stream.

You can use `throw` in your own code, instead of `error` when you want to be able to trap the error at a higher level, and can recover without causing the nuclear option.

To catch Exceptions, you use the `try / catch` block as follows; `finally` is optional, and will be explained below:

```
try
// words that might throw
catch // ( -- errmsg:s )
// your exception handling logic
finally // optional
// code that will always execute
/try
```

`try` signifies the beginning of an exception handling block. `try` blocks can be nested, both within the same word, as well as being nested by way of outer or inner words that may also have `try` blocks. In normal processing, the code between `try` and `catch` executes; the `catch` clause is skipped over, and if present, the `finally` clause executes.

If a word inside the `try` block, or any word it invokes, throws an exception via `throw`, then the following happens:

- the returns from every nested word are popped off of RS, to get the VM cleanly back up to the nesting level of the `try` block
- the data stack and all auxilliary stacks have their content restored to what it was at the entry to `try`
- the code in the `try` block stops executing
- the code in the `catch` block executes
- if the `catch` block does not then throw its own exception, the `finally` clause, if present, executes
- normal execution then resumes after the `/try` statement

If a `throw` occurs in runtime code, and there is no `try` block active at that time, the situation is handled just like `panic`.

How to use (and not use) exceptions and handlers

When to use throw

Use `throw` for unexpected error conditions, typically those cause by invalid externally supplied data, over which you had no control. It is especially indicated in operations

involving multiple steps, or that may become deeply nested.

Document in a word that has `throw`, and outer words that invoke it, that it throws. That provides some guidance that an exception handling block when invoking that operation may be helpful.

Don't use throw...

- as a new “short-circuit” channel for communicating results; `throw` is about exceptional error situations, not normal processing situations.
- as a replacement for having clear, sensible, understandable input requirements; there is not much point in using `throw` for programmer error; that deserves an `error` (or implicit runtime `panic`)

When to use try/catch

Don't think you need to include a `try` block just because something you are calling can `throw`; the question you need to ask, is whether there is anything reasonable you can do at the point. In many cases there may be, such as sending an error result on a result pipe, or returning a modal `false` result to a caller. But if there is nothing you can do, just let the exception continue up the chain, where something else might catch it, or in the worst case, the system will have to treat it as a panic.

Important Caveats and Limitations

SEH in a stack-based language is not as deterministic as in a procedural language. If your `try` block either consumes values from any stacks, or in normal execution leaves new or altered values on any stack, then this will likely NOT be the case if execution falls through to after `/try`. Therefore, you should base your handling logic on this fact.

Typically, you might want your `try` block to span your entire word, or at least be the last code in the word. If your word follows a modal (? – result true | false) pattern, your `catch` block can drop any input values on eg. DS that were restored by the handler, then push a `false` and fall thru.

As with *all* languages that support SEH, the side effects caused by intermediate words of your normal chain are in an indefinite state from your viewpoint, since you may not know how much executed prior to the `throw`.

Writing Clean Final Code

Stack-based languages like Final have unique beneficial qualities, but also come with hazards and potential pitfalls.

Because Final uses stacks and stack manipulation operators, it is missing the symbolic anchors of the named parameters of procedural languages. And due to these stack manipulations, it is also fairly easy to create defective code, that does these potential things:

- underflow a stack: use or discard a value under the intended current context;
- leak values on a stack, failing to drop a no-longer needed value;
- confuse the order of operands;
- mistake a stack value for something it isn't.

Use Local Variables

The best way to avoid stack confusion and bugs is to use [local variables](#). A local variable provides a simple mnemonic way to refer to parameters or subparameters, without having to remember what is currently on each stack.

Follow these guidelines

- consider moving stack-provided parameters to local variables, unless they are consumed only once;
- if the method is an object method, consider pulling fields you will access repeatedly from your `o`: instance and put them in a local;
- you don't have to define all your locals at the top, it can be clearer to define them "just in time" ahead of the code that needs them;
- if your subwords also access locals, you might consider defining those in a preliminary `init` subword, so they are defined when reading the the subword code. (Note: locals are resolved when used, and do not need to be lexically defined prior to use, so subwords are allowed to use locals that are only defined later in the file, either by other subwords, or the main word code.)

Factor!

Final enables you to factor your code in two ways:

- by defining conventional Final words, and delegating work to them;
- using a [subword](#) within a word.

Break long stretches of code, that has identifiable stages, into either subwords or another colon word.

“Clean coding” for procedural languages also recommends factoring larger procedures into small ones to which the main procedure delegates. But this often entails a fair bit of runtime overhead, insofar as you typically need to pass in a variety of parameters from the parent context to the delegate. At runtime, this is not cheap, since the runtime has to create new stack frames, copy parameter values, and so on.

Final by contrast enables much lower overhead factoring, especially for subwords. Since the values needed for an operation are already on the stacks, or also in locals for subwords, factoring is mostly a process of subdividing the work, and naming it clearly.

Factoring also lets you successively simplify your stack contexts, to narrow the values to just what a single step requires. This makes the factored code much easier to reason about, and to get correct.

General Guidelines

Benchmarking has demonstrated that Final code runs very efficiently on modern CPUs. Because Final has a limited quantity of underlying machine code, its machine code tends to stay hot in local CPU caches, which provides a comfortable cushion to write the safest, most understandable code, not worrying about saving an operation here and there.

Another example, is code that moves an object to eg TS, for receiver operations, and that will need to be dropped at the end. Sometimes these operations have a pattern like:

```
// ( obj:a -- )
>ts
  t:peek /...
  t:peek /...
  // last operations, which do i choose?
  ts> / ...
  // or:
  t:peek
  t:drop
```

You can save one instruction by moving the value on ts to ds on the last operation. But writing the code like the second example makes all stanzas identical, thus easy to

reason about, and dropping `ts.tos` is an explicit end step, closing the `>ts`. The second is the far better form, because it makes every stanza identical, thus the reader (which will be you!) sees that every stanza does the same thing, that the last stanza is not some exception to the workflow, and the stack hygiene is separated from the logic of the operations.

In the last example, I also demonstrated a helpful convention, which is, where possible, to nest code relating to the content of an auxiliary stack, opening and closing with the value going on then off the stack. This provides clear visual highlighting that all the nested code is working from the same value on eg `ts`.

Final Programming Conventions

Object operations that return a value, such as property reads, list item read, etc. do so on `ds`. If you start an attempted sequence of such operations against a single object, such as reading several properties to perform a check or perform an operation, you immediately hit a problem if you start with the object on `ds`. After the first read, the result is now `ds.tos`, so the instance is no longer available for the next operation. Rather than engaging in interminable `ds` juggling, which gets ever-more complex the more values you need to read, you should move the object to another stack, and root your operations there.

Rooted compound object operations always preserve the root object, so you never have to manually `dup` before a rooted operation. This makes the auxiliary stacks more convenient, and understable (writing and reading code).

Here are the stack usage guidelines that Final code follows, and that are suggested for user code as well. These guidelines suggest when and how to use auxiliary stacks, and how to make the choice of which stacks to use, and in what order. It should be noted that correct code operation does *NOT* depend on following these guidelines; all legal code that references the correct objects for its implementation will work; the guidelines just help provide semantic structure and familiarity of patterns.

Do continue to follow the Forth-like pattern of accepting the input parameters on `ds`, and placing the output parameters on `ds` as well, consuming the input parameters.

Modal Result-shape Operations

For operations that either can return a result or not, depending on factors determined by that operation, provide two return effects:

- when the operation succeeds: (? – result true)
- when the operation could not provide the result: (? – false) This pattern lets the consumer code be structured naturally, with a single test, whose false branch (`not if`) doesn't try to use the non-existent result, but whose true branch consumes the result.
- you typically mark this pattern by using ? affix in the name

If the result is truthy when it succeeds, but falsey when it doesn't, you can use this abbreviated pattern, along with ?dup that only duplicates a truthy result. This is suitable for a result that is an Int or String, where 0 or "" signal failure, but <>0 or <>"" signal success:

- when operation succeeds or fails, return the truthy or falsey result: (? – result) then you use code like this:

```
operation ?dup if
  // use the result, it was duped
else
  // no value to drop
then
```

For operations that don't produce a result, but need to signal an ok or an error, it is suggested to use the following pattern:

- when it succeeds: (? – true);
- when it fails: (? – err false).

You can use the `nif` variant of `if` in these situations

```
operation nif
  ( err )
  // handle error
then
```

Auxilliary Stack usage Conventions

Although all auxilliary stacks are identical in operation, by convention, ts is thought of as being the “Temp Stack”, and xyz as “parking” stacks.

So ts is your first goto for a short sequence of operations that will typically only involve a single object operand. For example:

```
// MyObj
//   .prefix
//   .name
//   .affix

: printFullName ( inst:o -- )
  >ts
  ( t:.prefix@ t:.name@ t:.affix@ )
  "%s%s%s" fmt $
  t:drop
```

Sometimes an operation may be a bit more complex, and involve one or more input operands, and then the fetching of materializing of one or more objects in its code. While it is always desirable to factor Final operations for the goals of readability, testability, or reusability, such factoring can itself involve careful design of the factor word, plus dealing with the potential problems of use by other code, which turns the factor word from an internal-intended part of the parent word into a public artifact, which could cause client use of the word to break if the parent word itself needs to refactor the api or semantics of the factor word. (A planned feature of nested words and local-only namespacing will help, when that lands.)

Other scenarios involve a top-level word used to perform some operation using one or more operand words, one or more internally sourced or materialized words, and output results.

All these types of words benefit from continuing to use `ts` for localized operations, with a beginning-middle-end, and then `x y` and `z` in a loose priority order, depending on need.

Auxiliary Stack Usage Heuristic

The basic heuristic, as mentioned, not enforced in any way, is to use the three stacks in reverse order of permanence or importance. So operations then dealing with a second object whose lifetime is longer than the use of `ts`, can use `xs`. A lifetime longer than `xs` can use `ys`. Etc.

In a situation where the top operation word is using or producing more than one word, and you want to keep those parked stably as you dispatch lower-level operations, you can follow a convention wherein input or source words are in order: `xs, ys, zs`; and outputs under construction parked in reverse order `zs ys xs`. Obviously this breaks down for more than three primary words, but rather than engaging in our unfriendly stack juggling, Final offers convenient help with Dynamic Objects.

If you are dealing with more state on input or output than you can reasonably juggle on txyz, you can easily create a `DynamicObject` to encapsulate the related items in a single object, which is then what you deal with, rather than disparate individual objects. This is also helpful to limit the number of parameters that need to be supplied on ds to an operation, or that are returned.

Word Doc Guidelines

The [Quick Tour](#) introduced **context effects**: a short comment that describes what a word expects on the stacks and what it leaves. This page is the full convention for writing those docs — the format the compiler stores, how `help` shows them, and how to document **Type members** (including your own [TypedObject](#) types).

Everyday API pages already point you at `help` for the same text (`help +` , `help Regex#find` , `help DateTime` , ...). Good docs make that useful.

Recommended comment form

Prefer a block comment with `/*` and `*/` **each on their own line**, and the body **flush left** (no indent inside the block):

```
/*
Add two numbers, and return the sum.
( a:n b:n -- c:n )
*/
: + ...
```

That is the form used throughout Final's own core docs (for example `+` in the basic-ops doc file). Single-line `// ...` comments also work for a one-liner, but the block form is clearer once you have more than a sentence.

Place the comment **immediately before** the word it documents (the colon definition, or a `:doc` line — see below). The compiler takes that **last comment** and stores it on the dictionary entry's `.doc` property. Typing `help wordname` in the UI prints that text.

What to write

Keep three layers, in order:

1. **Opening sentence** — one complete sentence that states the purpose of the word. Lead with the verb or outcome ("Add...", "True if...", "Create a new empty..."). Don't repeat the word name.
2. **Additional detail** — more complete sentences only when needed (panics, modal results, related words, nesting rules). Prefer short lines; skip anything the effect line already says.

3. **Last line** — the **context / stack effect**. Put nothing after it inside the comment.

Be succinct. Help text is read in the UI; long essays belong in these intro pages, not in `.doc`.

Good pattern

```
/*
Divide a by b and return the result.
Panics on divide by zero for integral types.
Returns Float.Inf for Float divide by zero.
( a:n b:n -- c:n )
*/
```

Modal results

When a word uses the success/failure shapes from [Writing Clean Final Code](#), spell both arms on the effect line:

```
/*
First match substring in s, or false if none. Modal.
A successful empty match is "" true (not false).
( s -- found:s true | false )
*/
```

Follow the same pattern if the word accepts a modal result.

Auxilliary stacks and the Object Stack

Bare `( ... -- ... )` is the **data stack**. When the word also uses or changes **ts / xs / ys / zs**, write those aspects with a letter prefix. Since the OS cannot be manipulated by user code, don't include before/after, only the one value that persists for the method:

```
/*
Render newline-separated ops onto the parked StringBuilder.
( text:s -- ) z( sb:StringBuilder -- sb )
*/

/*
Final method body: receiver is on OS for the duration of the call.
( -- s ) o( inst:Person )
*/
```


Only document aspects that change; omit stacks that are untouched, unless you want to emphasize they have a value that must flow through to code called in the word. In that case, you can use the flowthru convention.

```
/*
Helper word for parsing.
( -- ) z( sb:StringBuilder )
*/
```

Stack-effect notation

Inside (... -- ...) :

- Left of -- — inputs (deepest first; **TOS is rightmost**).
- Right of -- — outputs after the word runs.
- Optional `name:type` labels (e.g. `a:n`, `list:List`).

Common type codes

Code	Meaning
<code>b</code> or <code>t</code>	Bool / Truthy (most Bool contexts take Truthy; Json needs Bool)
<code>i</code>	Int
<code>u</code>	Uint
<code>iu</code>	Integral (Int or Uint)
<code>f</code>	Float
<code>n</code>	Numeric (Int, Uint, or Float)
<code>s</code>	String
<code>a</code>	Any
<code>DO</code>	DynamicObject (common shortcut)

For other types, write the full type name (`List` , `Regex` , `HttpReq` , ...).

Attaching docs to your TypedObject with :doc

Final provides the `:doc` helper to attach documentation to an existing word or Type. You cannot attach such docs to sealed core or module words, but you can document your own TypedObjects this way.

Syntax:

```
/*
...doc text...
*/
:doc wordName // e.g. your new TypeDef

/*
...doc text...
*/
:doc TypeName#memberName // for your instance and class properties, if you
want to document them
```

`:doc` reads the next identifier and attaches the **last comment** to that target's `.doc`. For members, use a single `#` between the type name and the member name — the same spelling you use in `help`:

Then, to refresh your memory on how your type works, you just need to type:

```
help MyType
```

Documenting type members

Core Final documents members this way (excerpt from the `maps/objects` doc file):

```
/*
True if the Map contains an entry for the key.
( key:a -- has:b )
*/
:doc Map#hasKey

/*
Create a new empty DynamicObject.
( -- o:DynamicObject )
*/
:doc DynamicObject#new
```

Your TypedObject types

If you define a `TypedObject` (see [TypedObject](#)), document its fields and Final methods the same way. After the type exists in the catalog:

```

/*
Person record: display name and age in years.
*/
:doc Person

/*
Display name.
*/
:doc Person#name

/*
Age in years.
*/
:doc Person#age

/*
One-line "Name (age)" description.
( -- s ) o( person -- person )
*/
:doc Person#describe

```

Prefer documenting members when you register them, so `help Person` and `help Person#describe` stay useful for anyone using your type — including yourself later.

You may still put a `/* ... */` block immediately above `: Person#describe` so the dictionary **Word** gets `.doc` automatically; use `:doc Person#describe` when you want the text on the **type member** that `help Type#member` and the type's member listing show.

Quick checklist

- `/*` and `*/` on their own lines; body flush left.
- First line: one complete purpose sentence.
- Extra sentences only for what the effect line cannot say.
- Last line: context/stack effect; nothing after it in the comment.
- Use `:doc name` or `:doc Type#member` when the target is not the word you just defined.
- Document TypedObject fields and methods so `help` works for your types.

Recursive Techniques

Final provides support for writing recursive code, and code that uses forward references. A good example is Final's own json parsing code. The words that parse individual json types, needs to be able to deal with any json type. For example, a json array can contain items of any json type.

This is re-entrant recursion, and is an appropriate technique when the depth is not expected to be too deep. (What is “too deep”? We'll get to limits later.)

Other times, an algorithm can be succinctly expressed as a step that subdivides or simplifies the work, then calls itself. Factorial is an often cited example.

Simple Recursion

During compilation of a Final word, the word itself is not yet visible in the dictionary. This presents a challenge in trying to use the word itself within itself. In this situation, Final provides a keyword, `recurse`; this has the same effect as naming the word itself; at runtime, it will nest into a new instance of the word.

```
// ( N:i -- N!:i )
: factorial
  dup 1 > if
    dup 1 - recurse *
  else
    drop 1
  then
;

```

When `N > 1`, the word keeps a copy of `N`, computes `(N-1)!` via `recurse`, then multiplies. For `0` and `1`, the base case drops the input and leaves `1`.

Forward References

Json parsing is a good example of re-entrant recursion. In this type of code, a word may not itself be recursive, but will indirectly call code that causes the nesting level of the overall algorithm to increase.

In this type of code, there is typically a chicken/egg in which earlier defined code must be able to call later-defined code. However Final, as it is defined, cannot directly support forward-references—a Word needs to be defined before it can be compiled.

In this case, Final provides `defer:`. It creates a word with no behavior yet. You assign the behavior later:

```
&of realWord is deferredWord
```

`&of` names the word that does the work. `is` makes that the behavior of the deferred word. You can compile calls to the deferred name before the real word exists. Until `is` runs, calling the deferred word panics.

You can also change that behavior later. A program can start with a basic version, then replace it once more of the system is defined.

Here is a short piece of Final's own Json parser. Arrays and objects both contain values, and a value may itself be an array or object, so `jsonParseArray` and `jsonParseObject` call `jsonParseValue` before that word's body is written:

```
// sole cycle: object/array members → value → object/array
defer: jsonParseValue

// ( -- list:List )
: jsonParseArray
  ...
  jsonParseValue    // may nest into another array or object
  ...
;

// ( -- obj:DynamicObject )
: jsonParseObject
  ...
  jsonParseValue
  ...
;

// ( -- v:a )
: (jsonParseValue)
  ...
  // ''' → string | '{' → object | '[' → array | ...
;

&of (jsonParseValue) is jsonParseValue
```

`defer: jsonParseValue` creates the name so the array and object parsers can call it. The real work is `(jsonParseValue)`. The last line assigns that word's behavior to `jsonParseValue`. Until that line runs, calling `jsonParseValue` panics.

Limits

Recursive and Re-entrant code can often provide a concise, easy to reason about algorithm. However these techniques are only suitable within limits of the level of recursion that happens at runtime. Final places depth limits on all its stacks, including the Return Stack. These limits are well above what almost any typical code would stress. But a recursive algorithm that results in excessive nesting can hit a data stack or return stack limit. In Final, that is a bit over 1,000.

So recursive/reentrant code needs to itself have limits. Recursion can always be replaced with iteration, so if the data set or recursion depth will exceed Final's limits, then it should be reworked using an iterated type of approach.

Output Formatting

`$` prints one value's natural string to stdout. `$$` prints one value's natural string to stderr. In `final-gui` and `final`, use **View ▶ Stdout/Stderr** to show the standing stream panes (Prompt stays); **View ▶ Prompt/History** returns to the Response and History panes.

When you need a composed message — several fields, fixed widths, decimals, hex, a quoted string — use `fmt` with a **List of values** and a **format string**.

You have already seen this pattern in [Writing Clean Final Code](#) and in object examples, e.g. `"%s (%d)" fmt . DateTime` also uses `( ... )` to build component Lists for factories. This page is the API detail for those sandwiches and for `fmt` itself.

API at a glance

Word	Effect	Description
<code>(</code>	<code>(--) r(- cm)</code>	open a value sandwich; mark current DS depth
<code>)</code>	<code>(...new... -- list:List)</code>	close sandwich → List (Any cells)
<code>s)</code>	<code>(...new... -- list:StringList)</code>	close sandwich → StringList (String cells only)
<code>fmt</code>	<code>(values:List fmt:s -- result:s)</code>	substitute values into the format string

Typical print pipeline:

```
( values... ) "format..." fmt $
```

`fmt` returns a **String** on DS. Print it with `$`, concatenate with `str::+`, or hand it to a [TextWriter](#) (`writeString`, `writeLine`).

The `( )` and `( s)` sandwiches

Both sandwiches share the same open word `(`. The closer chooses the list type:

Sandwich	Closer	Result	Typical use
<code>(...)</code>	<code>)</code>	List	<code>fmt</code> , <code>DateTime/Interval</code> factories, mixed values
<code>(... s)</code>	<code>s)</code>	StringList	<code>str::join</code> , string-only lists

How they work

1. `(` records the current Data Stack depth (a context mark on the return stack). It does **not** push a value.
2. Between `(` and the closer, push the items you want in the list — literals, property reads, method results, anything that leaves a value on DS.
3. `)` (or `s)`) gathers **only the values pushed after** `(`, packs them into a list in **writing order**, and leaves that list on DS.

List order matches left-to-right source order: the first value after `(` is `list[0]`; the last (TOS before the closer) is `list[n-1]`. That is also the order `fmt` walks for successive `%` markers.

```
( 10 20 30 )           // List: 10, 20, 30
( "a" "b" "c" s)       // StringList: "a", "b", "c"
```

`s)` requires every gathered cell to be a **String** (non-strings panic). Use `)` when cells may be `Int`, `Float`, `Bool`, or mixed.

Empty sandwiches are legal: `( )` → empty List; `( s)` → empty StringList.

You can also build a List with `List.new^` / `.append^` and pass that to `fmt`. The sandwich is the concise form for a fixed handful of values.

Only *new* values — a common trip-up

The sandwich does not include values that were already on DS when `(` ran. It collects solely the *new* cells placed on DS after the open.

```
"Ada" 36
( ) "%s (%d)" fmt $      // WRONG: empty list – Ada and 36 sit *below* the
mark
```



```
"Ada" 36
( "Ada" 36 ) "%s (%d)" fmt $ // works, but duplicates the literals
```

If the values you need are already on DS (or live on an object you are using), **do not expect) to “reach under” the mark**. Produce them *inside* the sandwich, or park them first and push fresh copies/reads after (.

Prefer parking on an auxiliary stack

Final’s aux stacks exist for this kind of problem ([Stack Operators](#), [Clean Final Code](#)). Park handles or operands, then emit values *inside* (...) :

```
: printFullName ( inst:o -- )
>ts
( t:.prefix@ t:.name@ t:.affix@ )
"%s%s%s" fmt $
t:drop
;
```

```
"Ada" >ts
36 >xs
( t:peek x:peek ) "%s is %d" fmt $
x:drop t:drop
```

Cautions

- Stack juggling after (to `pick / over` values from *below* the mark can work (a copied cell is “new”), but it is hard to read and easy to get wrong — park instead.
- Ops inside the sandwich that drop or rearrange cells *below* the mark can corrupt the caller’s stack. Keep the sandwich body a straight sequence of pushes (and rooted reads that leave results on DS).
- `s)` will reject non-String cells; if you need mixed types for `fmt`, use `)`.

fmt

```
// ( values:List fmtString:s -- result:s )
( 42 "widgets" ) "%d %s" fmt $ nl // 42 widgets
```

Rules of thumb:

- One value in the List per `%` substitution (not counting `%%`).

- Prefer an explicit type marker (`%d` , `%s` , ...) over a catch-all when you know the type.
- A count mismatch does not always abort the word; missing or extra substitutions show up as error text inside the result. Keep the List and the format string in sync.

`fmt` is for **building a String**. It does not write to stdout by itself — follow with `$` or a writer method when you want output.

Format codes (cheat sheet)

Format strings are ordinary Final Strings. Substitutions use `%` markers (C-style). Literal text copies through unchanged. Use `%%` for a literal percent sign.

Type markers (Final-relevant)

Code	Expects	Result
<code>%v</code>	any	default rendering of the value
<code>%s</code>	String	string contents
<code>%d</code>	Int (also fine for many integral values)	decimal integer
<code>%f</code>	Float	decimal floating point
<code>%t</code>	Bool	<code>true</code> / <code>false</code>
<code>%c</code>	Int (Unicode code point)	one character
<code>%x</code> / <code>%X</code>	Int	hex (lower / upper)
<code>%o</code>	Int	octal
<code>%b</code>	Int	binary
<code>%e</code> / <code>%E</code>	Float	scientific notation
<code>%g</code> / <code>%G</code>	Float	compact float (decimal or scientific)
<code>%q</code>	String	double-quoted string with escapes
<code>%%</code>	—	literal <code>%</code>

`%v` is a reasonable default when exploring. For production messages, prefer the typed codes so `Int/Float/Bool/String` stay obvious at the call site.

`%c` wants an **Int** code point (same idea as `r>s $` — interpret the `Int` as a character), not a one-character `String`. Prefer `%c` inside `fmt`; use `r>s` when you need a `String` for `$` / `$$` or other `String` APIs.

Width and precision

Insert width and precision between `%` and the type letter:

Form	Meaning	Example	Sample result
<code>%wd</code>	minimum width <code>w</code> , pad with spaces on the left	<code>(42) "%5d"</code> <code>fmt</code>	<code>" 42"</code>
<code>%0wd</code>	pad with leading zeros	<code>(42) "%05d"</code> <code>fmt</code>	<code>"00042"</code>
<code>%-ws</code>	width <code>w</code> , pad on the right (left-align)	<code>("ab") "%-6s"</code> <code>fmt</code>	<code>"ab "</code>
<code>%.pf</code>	precision <code>p</code> (digits after decimal for <code>%f</code>)	<code>(3.14159)</code> <code>"%.2f" fmt</code>	<code>"3.14"</code>
<code>%w.pf</code>	width and precision together	<code>(3.14159)</code> <code>"%8.2f" fmt</code>	<code>" 3.14"</code>
<code>%+.1f</code>	always show a sign	<code>(1.5) "%+.1f"</code> <code>fmt</code>	<code>" +1.5"</code>

Width applies similarly to `%s`, `%d`, `%x`, and friends. Precision for `%s` caps the number of characters taken from the string; for integers it is less often useful than for floats.

```
( "n" 7 3.5 ) "%s=%02d rate=%.1f%" fmt $ nl
// n=07 rate=3.5%
```

DateTime and Interval

`fmt` is for **general** messages (`%d`, `%s`, width/precision). It is **not** the `DateTime/Interval` layout API.

To format a **DateTime** or **Interval** with a letter-run pattern (or a curated constant such as `DateTime.iso` / `Interval.clock`), use **dt::fmt**:

```
DateTime.now^ DateTime.iso dt::fmt $ nl
5 Interval.minute dt::* Interval.clock dt::fmt $ nl
```

Full pattern tables, cascade rules, sign handling, and format constants: [DateTime and Interval — Formatting](#).

Examples

Several fields, then print

```
( "sensors" 3 true ) "%s: count=%d enabled=%t\n" fmt $
```

Single value still needs a List

```
( Int.maxInt ) "%d" fmt $ nl
( 255 ) "0x%02X" fmt $ nl
```

Join strings with (s)

```
( "alpha" "beta" "gamma" s ) ", " str::join $ nl
// alpha, beta, gamma
```

(`str::join` is covered under [String Operators](#); (s) is the handy way to hand it a `StringList`.)

Write into a builder / file

```
StringBuilder.new^ >ts
( "maxInt" Int.maxInt ) "%s=%d" fmt t:.writeString^
t:.toString^ $ nl
t:drop
```

Notes

- **\$ vs fmt** — `$` is one-shot natural print. `fmt` builds a `String` you can print, store, or send elsewhere.

- **Objects** — prefer pulling primitive fields (with rooted reads) into the sandwich rather than formatting a whole compound with `%v`.
- **Related** — `stdin/out Streams` for `$ / $$ / r>s`; `StringBuilder` and `TextWriter` for sinks; `DateTime` for `( parts )` factory Lists and for `dt::fmt` pattern formatting.

Input/UI Dialogs

When your Final code runs inside `final` or `final-gui`, you can raise modal dialogs that pause the script until the user chooses **Ok** or **Cancel**.

These APIs live on the class-only type `Dialog`. The mechanisms used to build and show these dialogs are the same for both `final` and `final-gui`. (You do not need to know anything about [Fyne and final-gui](#).)

There are three kinds:

Kind	When to use	Entry
Prompt	Yes/no confirmation with a short message	Simple Prompt Dialogs
Ask	Collect a handful of string responses, using a simple API	Simple Ask Dialog
Field	Collect a handful of typed values (name, flag, number, ...) using a more elaborated API than <code>ask</code>	Field Dialogs

All three are **blocking**: the call does not return until the dialog closes. Esc (and Break / Ctrl-B) means Cancel. Only one dialog may be open at a time.

You already know the pieces these APIs lean on:

- **Class methods** with `^` — [Methods](#)
- **Class constants** such as `Dialog.btnOk` — [Properties](#)
- **fields** `DynamicObjects` for Field Dialogs — [DynamicObject](#)
- The **modal result** shape `( ... -- result true \| false )` for Field Dialog Ok vs Cancel — [Writing Clean Final Code](#)

The next two pages cover the exact stack effects, parameters, and short worked examples.

Simple Prompt Dialogs

A **prompt dialog** shows a short message with **Ok** and **Cancel**. Use it when you only need a confirmation, not typed input.

API

Member	Kind	Effect	Description
<code>Dialog.showDefaultPrompt</code>	class method	<pre>(body:s -- btn:i)</pre>	show modal; return which button closed it
<code>Dialog.btnOk</code>	class const	<pre>(-- 1)</pre>	Ok was chosen
<code>Dialog.btnCancel</code>	class const	<pre>(-- 0)</pre>	Cancel was chosen (also Esc / Break)

`showDefaultPrompt` always uses title `"Prompt"` and focuses **Ok** by default. You pass only the body string. The body is shown centered; keep it short (about three lines).

Compare the returned `Int` to `Dialog.btnOk` / `Dialog.btnCancel` (or use `==` / `<>` against those constants).

Example

```
> "Overwrite notes.txt?" Dialog.showDefaultPrompt^
Dialog.btnOk == if
  "overwriting..." $ nl
else
  "left alone" $ nl
then

overwriting..
```

(In the run above, the user pressed **Ok**. Cancel or Esc would print `left alone`.)

A minimal check that only cares about Ok:

```
> "Continue?" Dialog.showDefaultPrompt^ Dialog.btnOk == $
true
```

Notes

1. The call **blocks** your Final word until the dialog closes; the rest of the UI (menus, Break) still works.
2. For a simple way to collect one or a few string fields, use an [Ask Dialog](#).
3. For collecting typed fields rather than a yes/no, use a [Field Dialog](#).

Simple Ask Dialogs

The Ask Dialog api provides a simple means for collecting from 1 to 7 string-typed values from the user, with a default prompt message.

The following example is in `examples/dialog/ask-demo.fnl`.

```
: ask-demo
( "What is your name?" "What is your quest?" )
Dialog.ask^ if
  "Nice to meet you, %s. Off you go - \"%s\" is a fine quest.\n" fmt $
else
  "Then we shan't proceed. Nobody expects the Spanish Inquisition.\n" $
then
;
```

The code provides the requested field labels using the convenient Paren operators for building a List. It then asks Dialog to post the dialog via the `ask` method, which returns one of two types of result shape:

- `true` then a List of the replies, in order of request
- `false` and no List

You test for the response type using an `if` statement, and if `true` can then pass the List directly to a `fmt` statement and print it out (or use the replies however otherwise). If `false`, you can print a default reply, as above.

`Dialog` is covered in more detail in [Input/UI Dialogs](#), Paren operators in [Compound Builders](#), and `fmt` in [Output Formatting](#).

Field Dialogs

A **field dialog** asks the user to fill in a small form and returns the edited values (or Cancel). `Dialog.newFieldDialogParams^` mints a params `DynamicObject` with defaults; you customize title, body, and the nested `fields` object, then call `Dialog.showFieldDialog^`.

API

Member	Kind	Effect	Description
<code>Dialog.newFieldDialogParams</code>	class method	<code>(-- params:DO)</code>	mint outer params with defaults and an empty <code>fields</code> object
<code>Dialog.showFieldDialog</code>	class method	<code>(params:DO -- fields:DO true false)</code>	validate, show modal; Ok → edited fields + true; Cancel → false only
<code>Dialog.btnOk / Dialog.btnCancel</code>	class const	<code>(-- i)</code>	same button constants as prompts; used for <code>defaultButton</code>

Outer params (after newFieldDialogParams)

Field	Type	Default	Role
<code>title</code>	String	<code>"Information Request"</code>	dialog title; must be non-empty if you set it
<code>bodyText</code>	String	<code>""</code>	instruction line above the fields; empty hides it
<code>defaultButton</code>	Int	<code>Dialog.btnOk</code>	which button is the default focus

Field	Type	Default	Role
			(<code>btnOk</code> or <code>btnCancel</code>)
<code>fields</code>	<code>DynamicObject</code>	empty object	the form rows; fill this before show

You may set `title`, `bodyText`, and `defaultButton` on the params DO, and add rows to `fields`, before calling `showFieldDialog`. The factory already created the nested `fields` object; park it (`t:.fields@ >xs`) and store your defaults there.

Fields object

- **1 to 8 fields**. An empty object panics at show time.
- **Creation order** is display order.
- Each **field name** is the label shown in the UI.
- The **default value's type** chooses the control:

Default type	Control	On Ok
String	full-width text line	edited String
Bool	checkbox	Bool
Int	narrow text	parsed Int (invalid → error, dialog stays open)
Float	narrow text	parsed Float (invalid → error, dialog stays open)

Any other default type panics when you show the dialog. User field names live only under `fields`, so they never clash with outer keys such as `title`.

On **Ok**, `showFieldDialog` pushes the **edited fields object** (not the outer params) and `true`. On **Cancel** / Esc / Break, it pushes only `false` — the modal [result shape](#) you already use elsewhere.

Example

Collect a small account form, then print what the user entered:

```

> Dialog.newFieldDialogParams^ >ts
"Account" t:.title!
"Enter name, age, and admin flag." t:.bodyText!
t:.fields@ >xs
"" x:.name!
0 x:.age!
false x:.admin!
x:drop
ts> Dialog.showFieldDialog^ not if
  "cancelled" $ nl
else
  >ts
  "name=" $ t:.name@ $ nl
  "age=" $ t:.age@ $ nl
  "admin=" $ t:.admin@ $ nl
  t:drop
then

name=Ada
age=36
admin=true

```

(Again, the printed lines assume the user filled those values and pressed **Ok.**)

A shorter pattern when defaults are enough and you only customize the title:

```

> Dialog.newFieldDialogParams^ >ts
"Connect" t:.title!
t:.fields@ >xs
"localhost" x:.host!
8080 x:.port!
x:drop
ts> Dialog.showFieldDialog^ not if
  "cancelled" $ nl
else
  >ts
  t:.host@ $ ":" $ t:.port@ $ nl
  t:drop
then

localhost:8080

```

Notes

1. Mint params with `Dialog.newFieldDialogParams^`, fill `title` / `bodyText` / `defaultButton` and the nested `fields` object, then `Dialog.showFieldDialog^`.

2. Int/Float rows stay open with a red error if the text does not parse; fix the field and press Ok again.
3. Same rules as prompts: one dialog at a time; Break cancels.
4. For a plain yes/no with no fields, prefer [Simple Prompt Dialogs](#).

Text and Byte Streams

Streams are sequential sources and sinks: you read or write **incrementally** (a rune, a line, a byte, a chunk) without loading everything at once. The same programming model applies whether the backing store is a file, an in-memory string or buffer, or the process's stdin/stdout.

Final splits streams along two axes you already saw in the [Features](#) survey:

Axis	Choices
Unit	Text (Utf-8 runes / strings / lines) vs bytes (raw octets in a ByteList)
Direction	Read (pull) vs Write (push)

That gives four **role** types — the shapes APIs ask for:

Type	Role
TextRead	read Utf-8 text
TextWrite	write Utf-8 text
ByteRead	read raw bytes
ByteWrite	write raw bytes

You usually open a concrete handle with a **class factory** on **TextReader**, **TextWriter**, **ByteReader**, or **ByteWriter** (`TextReader.newFromFile^`, `ByteWriter.newFromBytes^`, and so on). Those handles implement the matching role. Other types can too — for example **StringBuilder** is a **TextWrite**.

Final does not expose a general “open File, then ask it for a stream” workflow for day-to-day I/O — the stream factories *are* the entry point. (A small **File** helper exists for checks like `fileExists?`; see [File Streams](#).)

Already covered elsewhere

- `$` prints to the session's stdout ([Hello Final](#)).
- **ByteList** is the compact unsigned 8-bit buffer used with byte streams ([Numeric Lists](#)).
- Pips inherit stdout/stderr; prefer `write*` on a **TextWrite** / **ByteWrite** for real I/O ([Pips](#)).

In this section

- [Stream Types](#) — common API shapes, then Text and Byte members
- [File Streams](#) — path-based factories and open modes
- [String Streams](#) — in-memory String / ByteList / StringBuilder
- [stdin/out Streams](#) — session stdin, stdout, stderr

Use `list-streams` then `help TextRead` / `help TextReader` (and the other type names) in the UI for the full member list and stack effects.

Stream Types

This page is the shared API for text and byte streams. **TextRead / TextWrite / ByteRead / ByteWrite** are the roles; **TextReader, TextWriter, ByteReader, and ByteWriter** are the usual concrete handles (factories below). Backing-specific openers (file path, in-memory string, ByteList) are illustrated in the following pages; the **read / write / close / flush** members are the same regardless of how you opened the stream.

Common patterns

Opening: bare vs ?

File-backed factories come in two forms (same convention as casts and

`File.readBytes / readBytes?`):

- **bare** — must succeed; panics on open failure (`( path:s -- stream )`)
- **?** — modal result for caller-handled failure

Modal shape (same as [Writing Clean Final Code](#)):

- success: (`... -- stream true`)
- failure: (`... -- err:String false`)

```
"notes.txt" TextReader.newFromFile?^ not if
  $ nl                                // err string
else
  >ts
  // ... use t: ...
  t:.close^
  t:drop
then
```

When open **must** succeed (system / mandatory path):

```
"notes.txt" TextReader.newFromFile^ >ts
```

Reading: value plus OK

Text and byte **next / line** reads use the Final modal apishape:

Family	Effect
Text	(-- rune:i true false) or (-- line:s true false)
Byte	(-- b:i true false)

true means you got a value; **false** means end of stream (no placeholder value).

Chunked `readBytes` is different: it returns a **ByteList** (possibly empty) and does **not** use a flag — empty means no more data (or a non-positive count).

Writing, flush, and close

- Writers buffer; call `flush` when you need data to reach the backing store before `close`.
- `close` releases the backing handle when the provider supports it (files do; many in-memory providers are no-ops). Safe to call when you are done.
- Put the stream on **ts** (or another aux stack) for a short session of method calls, then `t:drop` — same style as other objects ([Writing Clean Final Code](#)).

Iteration

TextReader is iterable. Put the reader on DS and use `foreach ... /foreach`; `i` is each Unicode code point as **Int**. The loop consumes the reader (see [foreach](#)).

TextReader

Read Utf-8 text as runes or lines. A **TextRead** (`getNextRune`); `readLine` is on this concrete type.

Factories (class)

Name	Effect	Description
<code>newFromFile</code>	(path:s -- r)	open path; panics on error
<code>newFromFile?</code>	(path:s -- r true err:s false)	open path for sequential text read
<code>newFromString</code>	(s:String -- r true err:s false)	read over an in-memory String

Instance methods

Name	Effect	Description
<code>getNextRune</code>	<code>(-- r:i true false)</code>	next Unicode code point
<code>readLine</code>	<code>(-- line:s true false)</code>	next line without the newline; strips a trailing <code>\r</code>
<code>close</code>	<code>(--)</code>	release the reader

```
"hi" TextReader.newFromString^ drop >ts
t:.getNextRune^ if
  r>s $           // prints h
then
t:.close^
t:drop
```

TextWriter

Write Utf-8 text (runes, strings, or lines). A **TextWrite**.

Factories (class)

Name	Effect	Description
<code>newFromFileAppendOrCreate</code>	<code>(path:s -- w)</code>	append; create if missing; panics
<code>newFromFileAppendOrCreate?</code>	<code>(path:s -- w true err:s false)</code>	append; create if missing
<code>newFromFileTruncateOrCreate</code>	<code>(path:s -- w)</code>	truncate or create; panics
<code>newFromFileTruncateOrCreate?</code>	<code>(path:s -- w true err:s false)</code>	truncate or create

StringBuilder is a **TextWrite**: call its `writeRune` / `writeString` / `writeLine` members (same shape as below), then take `.toString^`. See [String Streams](#).

Instance methods

Name	Effect	Description
<code>writeRune</code>	<code>(r:i --)</code>	write one Unicode code point
<code>writeString</code>	<code>(s:String --)</code>	write as-is (no added newline)
<code>writeLine</code>	<code>(s:String --)</code>	write string then <code>\n</code>
<code>flush</code>	<code>(--)</code>	push buffered output
<code>close</code>	<code>(--)</code>	flush/release as needed

```
"out.txt" TextWriter.newFromFileTruncateOrCreate^ >ts
"hello" t:.writeLine^
t:.flush^
t:.close^
t:drop
```

ByteReader

Read raw bytes. A **ByteRead**. Values on DS are **Int** (0...255) for a single byte, or a **ByteList** for a chunk.

Factories (class)

Name	Effect	Description
<code>newFromFile</code>	<code>(path:s -- r)</code>	open path; panics on error
<code>newFromFile?</code>	<code>(path:s -- r true err:s false)</code>	open path for sequential byte read
<code>newFromBytes</code>	<code>(bytes:ByteList -- r true err:s false)</code>	read over an existing buffer

Instance methods

Name	Effect	Description
<code>getNextByte</code>	<code>(-- b:i true false)</code>	next octet
<code>readBytes</code>	<code>(n:i -- bytes:ByteList)</code>	up to <code>n</code> bytes (may be shorter / empty)
<code>close</code>	<code>(--)</code>	release the reader

```

ByteList.new^ >xs
65 x:.append^
66 x:.append^
x:peek ByteReader.newFromBytes^ drop >ts
t:.getNextByte^ if
  $                                // 65
then
t:.close^
t:drop
x:drop

```

ByteWriter

Write raw bytes into a file or append into a **ByteList**. A **ByteWrite**.

Factories (class)

Name	Effect	Description
<code>newFromFileAppendOrCreate</code>	<code>(path:s -- w)</code>	append; create if missing; panics
<code>newFromFileAppendOrCreate?</code>	<code>(path:s -- w true err:s false)</code>	append; create if missing
<code>newFromFileTruncateOrCreate</code>	<code>(path:s -- w)</code>	truncate or create; panics
<code>newFromFileTruncateOrCreate?</code>	<code>(path:s -- w true err:s false)</code>	truncate or create
<code>newFromBytes</code>	<code>(bytes:ByteList - w true err:s false)</code>	append writes into that list

Instance methods

Name	Effect	Description
<code>writeNextByte</code>	<code>(b:iu --)</code>	write one octet (Int or Uint)
<code>writeBytes</code>	<code>(bytes:ByteList --)</code>	write all bytes from the list
<code>flush</code>	<code>(--)</code>	push buffered output
<code>close</code>	<code>(--)</code>	flush/release as needed

```
ByteList.new^ >xs
x:peek ByteWriter.newFromBytes^ drop >ts
65 t:.writeNextByte^
ByteList.new^ >ys
66 y:.append^
67 y:.append^
y:peek t:.writeBytes^
y:drop
t:.flush^
t:.close^
t:drop
x:.len@ $ nl          // 3
x:drop
```

File Streams

File streams are opened by **class factories** on **TextReader**, **TextWriter**, **ByteReader**, and **ByteWriter**. Each file factory has two forms:

- **bare** — must succeed; panics on open failure `(( path:s -- stream ))`
- **?** — modal result `(( path:s -- stream true | err:s false ))`

Everyday sequential I/O goes through these factories. **File** also has whole-file `readBytes` / `writeBytes` / `readText` / `writeText` (same bare / ? pairing).

Text from a file

Factory	Effect	Opens
<code>TextReader.newFromFile</code>	<code>((path:s - - r))</code>	read-only; panics on error
<code>TextReader.newFromFile?</code>	<code>((path:s - - r true err:s false))</code>	read-only
<code>TextWriter.newFromFileAppendOrCreate</code>	<code>((path:s - - w))</code>	append; create if missing; panics
<code>TextWriter.newFromFileAppendOrCreate?</code>	<code>((path:s - - w true err:s false))</code>	append; create if missing
<code>TextWriter.newFromFileTruncateOrCreate</code>	<code>((path:s - - w))</code>	truncate or create; panics
<code>TextWriter.newFromFileTruncateOrCreate?</code>	<code>((path:s - - w true err:s false))</code>	truncate or create

```

"log.txt" TextWriter.newFromFileAppendOrCreate?^ not if
$ nl
else
>ts
"another line" t:.writeLine^
t:.flush^
t:.close^
t:drop
then

```

```

: printFileLines ( path:s -- )
  TextReader.newFromFile?^ not if
    $ nl
  else
    >ts
    forever
      t:.readLine^ not if
        break
      then
        $ nl
    /forever
    t:.close^
    t:drop
  then
;
"log.txt" printFileLines

```

`readLine` returns the line **without** the terminating newline. A trailing `\r` (CRLF files) is stripped. A final line that has no newline is still returned once; the next call returns **false**.

Bytes from a file

Factory	Effect	Opens
<code>ByteReader.newFromFile</code>	<code>(path:s - - r)</code>	read-only; panics on error
<code>ByteReader.newFromFile?</code>	<code>(path:s - - r true err:s false)</code>	read-only
<code>ByteWriter.newFromFileAppendOrCreate</code>	<code>(path:s - - w)</code>	append; create if

Factory	Effect	Opens
		missing; panics
<code>ByteWriter.newFromFileAppendOrCreate?</code>	<code>(path:s - - w true err:s false)</code>	append; create if missing
<code>ByteWriter.newFromFileTruncateOrCreate</code>	<code>(path:s - - w)</code>	truncate or create; panics
<code>ByteWriter.newFromFileTruncateOrCreate?</code>	<code>(path:s - - w true err:s false)</code>	truncate or create

```
"data.bin" ByteWriter.newFromFileTruncateOrCreate^ >ts
ByteList.new^ >xs
0 x:.append^
255 x:.append^
x:peek t:.writeBytes^
x:drop
t:.flush^
t:.close^
t:drop
```

```
"data.bin" ByteReader.newFromFile^ >ts
4 t:.readBytes^ >xs          // up to 4 bytes as ByteList
x:.len@ $ nl
x:drop
t:.close^
t:drop
```

Choose **Text*** when the file is Utf-8 text you want as runes or lines. Choose **Byte*** for opaque binary (images, protocols, hashes). Mixing layers in one handle is not how these APIs are meant to be used — open the unit you need.

Whole-file helpers on File

Method	Effect
<code>File.readBytes</code> / <code>readBytes?</code>	entire file → <code>ByteList</code>
<code>File.writeBytes</code> / <code>writeBytes?</code>	create/truncate; write a <code>ByteList</code>

Method	Effect
<code>File.readText</code> / <code>readText?</code>	entire file → <code>String</code>
<code>File.writeText</code> / <code>writeText?</code>	create/truncate; write UTF-8 of a <code>String</code>

Bare forms panic on I/O error; `?` forms use `( ... -- val true | err:s false )` (writes omit `val`).

File existence

```
// ( path:s -- b )
"notes.txt" File.fileExists?^ $ nl
```

Unexpected filesystem errors from `fileExists?` are treated as faults (they panic), not as a modal `false`. Missing paths return `false`.

Security note

File open factories and whole-file read/write are gated by session **Security Restrictions** (read vs write). Restricted Sessions used for embedding or AI tools may refuse file access. Details: [Security Restrictions](#).

String Streams

In-memory streams let you use the same **Text** / **Byte** APIs without touching the filesystem — handy for tests, parsing buffers you already hold, and building text before a single print or send.

Reading a String as text

```
// ( s:String -- r true | err:s false )
  TextReader.newFromString^
```

```
: printRunes ( s:String -- )
  TextReader.newFromString^ drop >ts
  t:peek foreach
    i r>s $
  /foreach
  t:.close^
  t:drop
;
"αβ\nγ" printRunes
```

For line-oriented work, keep the reader on **ts** and call `.readLine^` in a loop (same pattern as [File Streams](#)).

Newlines in the source String are ordinary runes (`\n`, and `\r` if present). `readLine` still treats `\n` as the line terminator and strips a preceding `\r`.

Writing text in memory: **StringBuilder**

StringBuilder is a **TextWrite** — the same write-shaped members as other text writers:

Name	Effect	Description
<code>new</code>	<code>(-- sb)</code>	empty builder
<code>writeRune</code>	<code>(r:i --)</code>	append a code point
<code>writeString</code>	<code>(s:String --)</code>	append as-is
<code>writeLine</code>	<code>(s:String --)</code>	append string then newline
<code>toString</code>	<code>(-- s:String)</code>	snapshot of accumulated text

```

StringBuilder.new^ >ts
"hello" t:.writeString^
32 t:.writeRune^
"world" t:.writeLine^
t:.toString^ $           // hello world\n
t:drop

```

You already saw this style under [Methods](#). Pretty / indented builders (`newWithPretty` , nesting helpers) are for structured rendering (for example Json); the plain builder is enough for most stream-like accumulation.

Bytes in memory: ByteList

ByteList is the buffer type for byte streams — a strongly typed list of unsigned 8-bit values ([Numeric Lists](#)).

Factory	Effect	Description
<code>ByteReader.newFromBytes</code>	<code>(bytes:ByteList -- r true err:s false)</code>	read sequentially from the list
<code>ByteWriter.newFromBytes</code>	<code>(bytes:ByteList -- w true err:s false)</code>	append writes into that same list

```

ByteList.new^ >xs
x:peek ByteWriter.newFromBytes^ drop >ts
72 t:.writeNextByte^      // 'H'
105 t:.writeNextByte^     // 'i'
t:.flush^
t:.close^
t:drop

x:peek ByteReader.newFromBytes^ drop >ts
2 t:.readBytes^ >ys
y:.len@ $ nl              // 2
y:drop
t:.close^
t:drop
x:drop

```

The writer **appends** to the list you passed in; it does not replace earlier contents. The reader walks from the start of the list with its own offset — it does not remove items from the **ByteList**.

stdin/out Streams

Every Final **Session** / run has three text streams bound for you:

Stream	Type	Role
stdin	TextRead (usually a TextReader)	input (REPL lines, scripted input)
stdout	TextWrite (usually a TextWriter)	ordinary program output
stderr	TextWrite (usually a TextWriter)	diagnostics and fault reports

You do **not** open these with the file or in-memory factories. The runtime wires them when the Session starts — process stdio for an interactive TUI or headless REPL, capture buffers when a host embeds Final, or null readers/writers when a worker has no input/output.

Everyday printing: \$ and \$\$

For most programs you never hold the stdout handle yourself. Words you already used write there for you:

Word	Effect	Description
\$	(val:a --)	print the value's natural string to stdout
\$\$	(val:a --)	same natural string, to stderr

```
> "Hello from Final!\n" $
```

```
Hello from Final!
```

```
> "warning" $$
```

Acceptable `$` / `$$` inputs include **String**, **Int**, **Uint**, **Float**, **Bool**, and **DateTime**. Other objects typically print only their type name — not a full dump. For richer formatting of several values, see [Output Formatting](#) (`fmt` and friends).

Printing a Rune

A **Rune** is a Unicode code point carried as an **Int**. Ordinary `$` on an `Int` prints the **decimal** code point (`65 $` → `65`), which is rarely what you want for text. Convert first with `r>s`, then print:

Word	Effect	Description
<code>r>s</code>	<code>(rune:i -- s)</code>	Int code point → one-character String

```
> 65 r>s $ 10 r>s $
```

```
A
```

When you are already building a `fmt` message, use `%c` with the `Int` code point instead — no `r>s` needed:

```
> ( 65 ) "char=%c" fmt $ nl
```

```
char=A
```

On a **TextWrite** handle, call `writeRune` directly; conversion is only for the `$` / `$$` path (and any other String sink).

When you care about the handle

- **Pips** copy the parent's stdout/stderr into the spawn parameter object by default; process stdout/stderr are safe for concurrent writes. Custom writers you create (files, builders) are **not** automatically locked — your program owns them. See [Pips](#).
- Worker pips get a **null** stdin unless you bind an explicit reader; line-oriented REPL input stays on the main pip.
- Prefer **TextWrite** / **ByteWrite** `write*` methods (and `fmt`) for anything beyond quick `$` prints. Do not treat stdout as a general mux for files or network traffic.

Null streams

When no real input or output is bound, reads EOF immediately and writes are discarded. That is normal for some embedded or worker setups — not an error by itself.

File System

Final's filesystem API covers **paths**, **metadata**, **directories**, and **whole-file** helpers. Sequential streaming I/O (line/byte readers and writers) lives under [File Streams](#) — this page does not repeat open/read/write stream factories.

You already know the pieces these APIs lean on:

- Class methods with `^` — [Methods](#)
- Modal `( ... -- result true \| err:s false )` — [Writing Clean Final Code](#)
- `foreach` over iterators — [Iterators: foreach](#)

Type	Role
Path	Class-only path-string algebra (join, split, clean, ...)
File	Exists / info / copy / move / delete; whole-file read/write; name helpers
Dir	Directory root value; cwd; mkdir; filesystem iteration
FileInfo	Metadata snapshot (name, size, modTime, isDir, ...)

Use `help File` / `help Dir` / `help Path` / `help FileInfo` in the TUI for the full member list.

Path (class-only)

Operands are **strings** on the data stack. There are no Path instances.

Method	Effect	Description
<code>join</code>	<code>(base:s name:s -- s)</code>	join with host separator
<code>split</code>	<code>(s -- list:StringList)</code>	path components (absolute keeps root)
<code>clean</code>	<code>(s -- s)</code>	drop redundant <code>.</code> / <code>..</code>
<code>abs</code>	<code>(s -- s)</code>	absolute path (uses process cwd)
<code>basename</code>	<code>(s -- s)</code>	leaf without extension
<code>dirname</code>	<code>(s -- s)</code>	parent directory

Method	Effect	Description
<code>extname</code>	<code>(s -- s)</code>	extension with leading <code>.</code> , or empty

```
"out" "report.md" Path.join^          // "out/report.md" (host sep)
"a/./b/./c" Path.clean^              // "a/c"
"out/reports/q1.md" Path.basename^    // "q1"
"out/reports/q1.md" Path.extname^     // ".md"
"/var/log/syslog" Path.split^ foreach i $ sp /foreach
```

FileInfo

Snapshot returned by `File.info` / `Dir.info` and by directory iteration. No public `new`.

Prop	Type	Description
<code>path</code>	String	path for this entry
<code>name</code>	String	leaf name
<code>size</code>	Int	size in bytes
<code>modTime</code>	DateTime	modification time
<code>isDir</code>	Bool	directory?
<code>isSymlink</code>	Bool	symlink?

File (class methods)

Most ops take a **path string**. Mutators use the modal shape unless noted.

Method	Effect	Description
<code>exists? / fileExists?</code>	<code>(path:s -- b)</code>	missing → <code>false</code>
<code>info</code>	<code>(path:s -- fi:FileInfo true err:s false)</code>	metadata
<code>delete</code>	<code>(path:s -- true err:s false)</code>	file or empty directory

Method	Effect	Description
<code>move</code>	<code>(src:s dst:s -- true err:s false)</code>	rename/relocate; dst is full path ; no clobber
<code>copy</code>	<code>(src:s dstdir:s -- true err:s false)</code>	copy into directory; keeps leaf name; no clobber
<code>getBaseExt</code>	<code>(name:s -- base:s ext:s)</code>	leaf split; ext without dot
<code>makeName</code>	<code>(base:s ext:s -- name:s)</code>	join base+ext; empty ext → base alone
<code>readText / readText?</code>	<code>(path:s -- s) / modal</code>	whole file as String
<code>writeText / writeText?</code>	<code>(path:s s --) / modal</code>	create/truncate UTF-8
<code>readBytes / readBytes?</code>	<code>(path:s -- bytes) / modal</code>	whole file as ByteList
<code>writeBytes / writeBytes?</code>	<code>(path:s bytes --) / modal</code>	create/truncate bytes

Bare read/write forms **panic** on I/O error; `?` forms are modal. Stream open/read/write: [File Streams](#).

```
"archive/report.final.md" File.getBaseExt^ // base="report.final"
ext="md"
"out/report" "md" File.makeName^ // "out/report.md"
"." File.exists?^
"." File.info^ if >ts t:.name@ $ sp t:.isDir@ $ nl t:drop else $ nl then

"tmp/notes.txt" "Apple" File.writeText?^
"tmp/notes.txt" "tmp/archive" File.copy^
"tmp/archive/notes.txt" "tmp/archive/notes-2.txt" File.move^
"tmp/archive/notes-2.txt" File.delete^
```

Dir

A **Dir** is a directory root value (path), not an open OS handle.

Kind	Name	Effect	Description
class	<code>new</code>	<code>(path:s -- dir)</code>	Dir for path
class	<code>cwd</code>	<code>(-- dir)</code>	process working directory
class	<code>setCwd</code>	<code>(path:s -- true err:s false)</code>	change process cwd
class	<code>exists?</code>	<code>(path:s -- b)</code>	path exists?
class	<code>info</code>	<code>(path:s -- fi true err:s false)</code>	metadata
class	<code>mkdir</code>	<code>(path:s -- true err:s false)</code>	<code>MkdirAll</code> ; idempotent if already a dir
prop	<code>string</code>	String	canonical path string for this Dir
method	<code>iter</code>	<code>(flags:i -- iter)</code>	<code>FileSystemIter</code> of <code>FileInfo</code> under this Dir

Iteration flags (class properties on Dir)

Combine with bitwise OR:

Flag	Meaning
<code>Dir.IterDirs</code>	include directories
<code>Dir.IterFiles</code>	include files (non-dirs)
<code>Dir.IterDeep</code>	recurse
<code>Dir.IterAll</code>	dirs + files + deep

```

"." Dir.new^ >ts
t:.string@ $ nl

"tmp/demo" Dir.mkdir^
"tmp/demo" Dir.new^ >ts
Dir.IterFiles t:.iter^
foreach
  i >xs
  x:.name@ $ nl
  x:drop
/foreach
t:drop

```

Deep walk (pick a **small** root — recursive walks can be large):

```
: fileSysIter ( root:s -- )
  Dir.new^ >ts
  Dir.IterAll t:.iter^
  foreach
    i >ts
    t:.isDir if ">" else " " then
      "%s %9d %v %s %s\n" fmt $
      drop
    t:drop
  /foreach
  t:drop
;

"." fileSysIter
```

Security

Filesystem members are gated by **SR.FileRead**, **SR.FileWrite**, and **SR.FileSystemRead** (exists/info/iter/cwd vs content open). Restricted Sessions may deny them. See [Security Restrictions](#).

Examples on disk

- `examples/file-system/path.fnl`
- `examples/file-system/file.fnl`
- `examples/file-system/dir.fnl`
- `examples/file-system/filesys-iter.fnl`

Regex

Regex is Final's sealed type for regular-expression matching. It wraps Go's RE2 engine (`regexp`): linear-time matching, no backreferences, and no lookahead. Prefer [String Operators](#) (`str::startsWith`, `str::indexOf`, ...) when an exact substring or fixed separator is enough; reach for `Regex` when the pattern itself varies.

The [Quick Tour](#) already flagged “compiled regex, matches, selectors.” [Strings](#) pointed here for pattern matching. This page is the API and the day-to-day patterns.

Type `help Regex` (or `help Regex#find`) in the TUI for the same stack effects inline.

When to use what

Approach	Use when	How
Exact string probes	Fixed prefix, suffix, or substring	<code>str::...</code> (ops)
One-shot Regex	A single check or extract; pattern is not reused	Class methods <code>Regex.match^</code> / <code>Regex.find^</code>
Compiled Regex	Same pattern over many subjects, or you need find-all / split / replace / captures	<code>Regex.new^</code> , park on ts , call instance methods

Stack order is **subject then tool**: the `String` is deeper; the pattern (or replacement) is TOS — same posture as other Final APIs.

Patterns are easiest as **raw strings** (backticks), so `\d` and friends need no double-escaping ([Interpreter](#)).

Invalid pattern text is a programming error: `Regex.new^` and the one-shot helpers **panic**. While drafting in the REPL, probe with `Regex.isValid?` first.

One-shot (class)

No `Regex` instance is left on the stack. Fine for a one-off validation or first-match extract.

Name	Effect	Description
<code>new</code>	<code>(pattern:s -- re:Regex)</code>	compile; panics if invalid
<code>isPatternValid?</code>	<code>(pattern:s -- b)</code>	<code>true</code> if the pattern would compile
<code>match</code>	<code>(s:String pattern:s -- b)</code>	<code>true</code> if the pattern matches somewhere in <code>s</code>
<code>find</code>	<code>(s:String pattern:s -- found:s true false)</code>	first match substring, or modal <code>false</code>
<code>quoteMeta</code>	<code>(s -- q:s)</code>	escape metacharacters so <code>s</code> matches as a literal

```

"42" `^\\d+$` Regex.match^ $ nl           // true

"hello 42 there" `\\d+` Regex.find^ not if
  "no digits" $ nl
else
  $ nl                                     // 42
then

`(unclosed` Regex.isPatternValid? $ nl     // false
`price $` Regex.quoteMeta $ nl            // price \$

```

`find` is modal ([Writing Clean Final Code](#)): success leaves the substring then `true`; failure leaves only `false`. A successful **empty** match is `"" true`, not `false`.

Compiled (instance)

Build once with `Regex.new^`, reuse on many subjects. Park the receiver on **ts** for a short session of method calls ([Methods](#), [Clean Final](#)).

Properties

Name	Type	Description
<code>pattern</code>	String	pattern text
<code>numGroups</code>	Int	number of capturing groups (not counting the whole match)

Name	Type	Description
<code>groupNames</code>	<code>StringList</code>	keys for groups in Go order: <code>"0"</code> (whole match), then <code>"1"</code> ... or names; length is <code>.numGroups@ + 1</code>

Instance methods

Name	Effect	Description
<code>match</code>	<code>(s -- b)</code>	<code>true</code> if this Regex matches somewhere in <code>s</code>
<code>find</code>	<code>(s -- found:s true false)</code>	first match substring (modal)
<code>findAll</code>	<code>(s -- parts:StringList true false)</code>	all non-overlapping matches (modal; <code>false</code> if none)
<code>findAllN</code>	<code>(s n:i -- parts:StringList true false)</code>	at most <code>n</code> matches (<code>n < 0 → all</code>)
<code>findIndex</code>	<code>(s -- start:i end:i true false)</code>	byte indexes of the first match (modal)
<code>split</code>	<code>(s -- parts:StringList)</code>	split on matches (always a list; one element if no split)
<code>splitN</code>	<code>(s n:i -- parts:StringList)</code>	split into at most <code>n</code> parts
<code>matchObject</code>	<code>(s -- o:DynamicObject true false)</code>	whole match + captures as a DO (modal)
<code>replaceAll</code>	<code>(s repl:s -- out:s)</code>	replace every match; <code>repl</code> may use <code>\$1</code> , <code>\$name</code> , <code>\$0</code> , ...
<code>replaceAllLiteral</code>	<code>(s repl:s -- out:s)</code>	replace every match; <code>repl</code> is literal (no <code>\$</code> expansion)

```

`\\w+` Regex.new^ >ts

"one two three" t:.findAll^ not if
  "no words" $ nl
else
  // StringList of matches on DS
  foreach
    i $ sp
  /foreach
    nl
  then

"a1b2c3" t:.findIndex^ not if
  "no match" $ nl
else
  // start end on DS (byte indexes)
  $ sp $ nl
  then

t:drop

```

Captures with matchObject

On success, fields follow standard numbering: `"0"` is the whole match; `"1"`, `"2"`, ... are capturing groups. Named groups use `(?P<name>...)` and appear under that name (still numbered as well). Creation order matches Go group indexes, so `k @()` is group `k`.

Access ([Collection Accessors](#), [DynamicObject](#)):

- named group — `.name@` (Final ident) or `"name" @{}`
- any key including `"0"` — `@{}` (string key)
- by position — `@()` with an Int
- `.0@` is **invalid** (`"0"` is not a Final identifier)

```
`(?P<code>[A-Z]+)-(P<num>\d+)` Regex.new^ >ts
"AB-42" t:.matchObject^ not if
  "no match" $ nl
else
  >xs
  x:.code@ $ sp          // AB
  x:.num@ $ nl          // 42
  "0" x:@{} $ nl        // AB-42 (whole match)
  1 x:@() $ nl          // AB
  x:drop
then
t:drop
```

Replace

```
`(\d+)` Regex.new^ >ts
"a12b34" "#$1" t:.replaceAll^ $ nl          // a#12b#34
t:drop

`\$` Regex.new^ >ts
"a$b" "USD" t:.replaceAllLiteral^ $ nl      // aUSDb
t:drop
```

Examples

Validate once (one-shot)

```
: isZip? ( s -- b )
  `^\d{5}(-\d{4})?$` Regex.match^
;

"80302" isZip? $ nl          // true
"80302-1234" isZip? $ nl     // true
"8030" isZip? $ nl           // false
```

Reuse a compiled pattern

```
`\d+` Regex.new^ >ts
"x=10 y=20" t:.findAll^ not if
  "none" $ nl
else
  foreach i $ sp /foreach nl
then
  "only 7 here" t:.find^ not if "none" $ nl else $ nl then
t:drop
```

Split on whitespace

Unlike `str::split` (exact separator), `Regex split` treats the pattern as the delimiter:

```
`\s+` Regex.new^ >ts
"alpha  beta\tgamma" t:.split^ >xs
x:.len@ $ nl // 3
0 x:@() $ nl // alpha
x:drop
t:drop
```

Pattern cheat sheet (RE2)

Final patterns are Go/RE2. There are **no** `\1` backreferences and **no** `(?=...)` / `(?!...)` lookahead. For the full grammar, see Go's [regexp/syntax](#) docs.

Characters and classes

Pattern	Meaning
<code>.</code>	any character except newline (unless <code>(?s)</code>)
<code>[abc]</code> / <code>[^abc]</code>	character class / negated
<code>[a-z0-9]</code>	ranges inside a class
<code>\d</code> <code>\D</code>	digit / non-digit (<code>[0-9]</code> / <code>[^0-9]</code>)
<code>\w</code> <code>\W</code>	word char <code>[0-9A-Za-z_]</code> / non-word
<code>\s</code> <code>\S</code>	whitespace / non-whitespace
<code>\p{Greek}</code> / <code>\P{Greek}</code>	Unicode property / negated
<code>[:alpha:]</code>	ASCII POSIX class (inside <code>[]</code>)

Anchors

Pattern	Meaning
<code>^</code> <code>\$</code>	start / end of text (also line starts/ends with <code>(?m)</code>)
<code>\A</code> <code>\z</code>	absolute start / end of text
<code>\b</code> <code>\B</code>	ASCII word boundary / non-boundary

Repetition

Pattern	Meaning
<code>x*</code> <code>x+</code> <code>x?</code>	0+, 1+, 0 or 1 (greedy)
<code>x*?</code> <code>x+?</code> <code>x??</code>	same, prefer fewer (non-greedy)
<code>x{n}</code> <code>x{n,}</code> <code>x{n,m}</code>	exactly n; n or more; n...m

Groups and alternation

Pattern	Meaning
<code>x y</code>	<code>x</code> or <code>y</code> (prefer left)
<code>(re)</code>	capturing group
<code>(?:re)</code>	non-capturing group
<code>(?P<name>re)</code>	named capturing group (use with <code>matchObject</code>)

Flags (inline)

Pattern	Meaning
<code>(?i)</code>	case-insensitive
<code>(?m)</code>	multi-line: <code>^</code> / <code>\$</code> also match line edges
<code>(?s)</code>	“dotall”: <code>.</code> matches newline
<code>(?U)</code>	ungreedy default for quantifiers
<code>(?flags:re)</code>	apply flags only to <code>re</code>

Literals and replace templates

Need	How
Metacharacter as literal	escape with <code>\</code> , or wrap text via <code>Regex.quoteMeta</code>
Whole-string match	anchor with <code>^...\$</code>

Need	How
<code>\$</code> in <code>replaceAll</code> <code>repl</code>	<code>\$1</code> , <code>\$name</code> , <code>\$0</code> expand; use <code>replaceAllLiteral</code> when <code>\$</code> must stay literal

Indexes from `findIndex` are **Utf-8 byte** offsets (same caution as `str::len` / `str::indexOf` — do not invent a byte-range substring API; use the `matchString` from `find` / `matchObject` instead).

Json

Final provides full support for rendering and parsing JSON, according to RFC 8259 and RFC 3339 for formatting DateTime values.

There is an entry below for the two main topic:

- [Json Rendering](#)
- [Json Parsing](#)

Rendering

The table below lists the Final types that are supported for Json rendering, and what Json type they render to. The entire string is in Utf-8 encoding.

Final type	Json type	Note
Null	null	
Bool	true / false	
Int	number	1
UInt	number	1
Float	number	2
String	string	3
List	array	4
Map, DynamicObject, TypedObject	object	5
DateTime	string	6

1. No decimal.
2. With a decimal.
3. `\t`, `\r`, and `\n` are escaped.
4. The type of each list value determines the resulting Json type
5. The type of each Map value determines the resulting Json type
6. DateTime is formatted in ISO format, per RFC 3339

Nesting of structured objects in Lists and Maps is fully supported.

There are 2 words in Final used to create a Json string:

Create a compact json result, no unnecessary spaces, no newlines:

```
// ( root:a -- json:s )
>json
```

Create a “pretty” json result, with nesting and newlines, uses two space indenting:

```
// ( root:a -- json:s )
>jsonp
```

“Pro tip” use `>jsonp $` in Final when you want a quick display of collection objects like Lists, Maps, and Dynamic/Typed Objects!

Examples

Primitives

```

null >json $ nl           // null
true >json $ nl           // true
-49 >json $ nl            // -49
3.14 >json $ nl           // 3.14
"hi\n" >json $ nl         // "hi\n"

```

List (array)

`( ... )` builds a temporary List on DS — convenient for small Json arrays:

```

( null true -49 "rain" ) >json $ nl
// [null,true,-49,"rain"]

( null true -49 "rain" ) >jsonp $

```

Pretty output uses two-space indent:

```

[
  null,
  true,
  -49,
  "rain"
]
```

DynamicObject (object)

```

DynamicObject.new^ >ts
null t:.Null!
true t:.Bool!
-49 t:.Int!
"hello" t:.msg!
ts>
>json $ nl
// {"Null":null,"Bool":true,"Int":-49,"msg":"hello"}

```

Field order follows **creation order** on the DynamicObject.

Nesting

Lists and objects nest freely:

```
: makePayload
  DynamicObject.new^ >ts
  ( 1 2 3 ) t:.nums!
  DynamicObject.new^ >ys
  "world" y:.hello!
  ys> t:.nested!
  ts>
;

makePayload >jsonp $
```

Typical pretty result:

```
{
  "nums": [
    1,
    2,
    3
  ],
  "nested": {
    "hello": "world"
  }
}
```

Reading/Parsing

Final can parse any legal Json. The table below shows the Final Types used for each Json type.

Json type	Final type	Note
null	Null	
bool	Bool	
number	Int or Float	1
string	String	2
array	List	3
object	DynamicObject	4

1. **Int** when the value has no decimal and fits in **Int**; otherwise **Float**.
2. All json escapes are processed, including hex Unicode
3. The value of each item will follow the Json->Final mapping; full nesting of arrays and objects is supported.
4. The value of each object value will follow the Json->Final mapping; full nesting of arrays and objects is supported.

The everyday parse word is:

```
// ( json:s -- root:a )
json>
```

Invalid Json **throw s** a string message rather than panicking. Wrap `json>` in `try / catch` when the text is untrusted (an HTTP body, a file, a tool argument). If nothing catches, an uncaught `throw` is handled like a panic. See [Error and Exception Handling](#).

There is also a lower-level `parseJson` with the same stack effect; prefer `json>` in application code.

Examples

Primitives

```
"null" json> $ nl      // null
"true" json> $ nl      // true
"-49" json> $ nl       // -49
"3.14" json> $ nl      // 3.14
`"hello"` json> $ nl   // hello
```

Array → List

```
`[1,2,3]` json> >ts
0 t:@() $ sp          // 1
1 t:@() $ sp          // 2
2 t:@() $ nl          // 3
t:drop
```

Or walk with `foreach` :

```
`["a","b","c"]` json>
foreach
  i $ sp
/foreach
nl
// a b c
```

Object → DynamicObject

```
`{"x":1,"y":2,"msg":"hi"}` json> >ts
t:.x@ $ sp            // 1
t:.y@ $ sp            // 2
t:.msg@ $ nl          // hi
t:drop
```

String-key access works the same as any `DynamicObject`:

```
`{"host":"wttr.in"}` json> >ts
"host" t:@{ } $ nl    // wttr.in
t:drop
```


Nesting and round-trip

```

: sampleJson
  `{"nums":[1,2,3],"nested":{"hello":"world"}}`
;

sampleJson json> >ts
t:.nums@ foreach i $ sp /foreach nl      // 1 2 3
t:.nested@ >ys
y:.hello@ $ nl                          // world
y:drop
ts> >jsonp $                            // pretty round-trip

```

A compact round-trip is just `json> >json`. Values that were numbers without a decimal come back as **Int** when they fit; otherwise **Float**.

Json Parsing Limits

The following limits are checked when parsing json, to protect against malicious or unreasonable json. If a limit is reached during parsing, `json>` **throws** a string stating the limit type and the limit value.

Limit	Value	Note
Input length	8,388,608	Maximum size of the JSON text (about 8 MB).
Nesting depth	64	How deeply arrays and objects may nest inside each other.
Node count	1,000,000	Maximum number of JSON values — objects, arrays, strings, numbers, true, false, and null. Field names are not counted. An empty object or array counts as one value.
Identifier length	2,048	Maximum length of an object field name.
String length	1,048,576	Maximum length of a string value (about 1 MB), after JSON escapes are decoded.
Number length	64	Maximum length of a number as written in the JSON (sign, digits, decimal point, and exponent).

Threading

Final provides first-class threading support using a highly memory efficient model. Many languages map a language thread directly to an Operating System thread. But OS threads are heavy beasts that require a lot of setup and teardown time, and consume scary amounts of memory per thread.

Final uses an M:N model, in which it maps lightweight language threads (called `Pips`) to a fixed pool of expensive operating system threads. It has a scheduler which allocates ready-to-run Pips to an available OS thread. This enables you to create threads in the thousands or even hundred of thousands if it makes sense. N is chosen by the runtime to map effectively to the CPU, and is typically equal to the number of cores of the host.

The preferred method of interthread communication (ITC) in Final is via Pipes, which are conduits that can be shared between threads, and the root, to send or retrieve values.

Although Final encourages a “communicate to share data” not “share data to communicate” model, it does offer some threadsafe Atomic objects that can be safely shared between threads. The iterable Lists are particularly useful, since they use an efficient form of snapshotting of items at iteration time, which enables such lists to iterate without taking a lock.

Read on to learn about `Pips` `Pipes` and `Atomics`!

Pips

A **Pip** is a Final virtual machine with its own stacks and instruction pointer, running on its own lightweight thread — independent of the pip that started it. Think of it as a sprightly little worker.

All pips in a Session share one dictionary context (dictionary, code table, and data table). Only the **main** pip may compile or define new words. Once any child pip is running, the dictionary is locked: compile and define attempts raise an error. A dictionary may also be sealed after a successful compile (for example, core after bootstrap); that is separate from the live-child lock.

Pips are lightweight — each needs only a small set of empty stacks to exist — so you can create many of them when that fits the work, especially for blocking I/O such as files or the network. Still, a machine has a limited number of cores that actually run at once. For CPU-bound “split the work” jobs, keep subdivision modest (for example, on the order of 16 pips), not hundreds.

Spawning

Spawning starts a word with a parameter object in a new Pip (thread). When that word returns, the pip exits. There is no join: you do not wait on a pip handle. Coordinate with [pipes](#) (and optionally shared atomics).

To spawn a new Pip (thread):

```
Pip.newPipParams^ spawn: doWork
```

spawn: reads the next identifier, checks that the word is suitable to spawn, and launches it. For lower-level use, **(spawnPip)** takes the parameter object and an execution token obtained with **&of**. Use **help spawn:** and **help (spawnPip)** for stack effects and details.

The word you spawn should be an ordinary colon word — not immediate, hidden, compile-only, or still being defined.

Note that the root Task in a Session runs under the Final Interpreter. Spawned words do not, and only execute the code in the spawned word.

PipParams

“PipParams” is a convention, not a separate type. `Pip.newPipParams^` builds an `AtomicDynamicObject` parameter object (threadsafe if the object is shared across pips) with:

Field	Default	Role
<code>_stdout</code>	current stdout (TextWrite)	Required at spawn — child’s stdout
<code>_stderr</code>	current stderr (TextWrite)	Required at spawn — child’s stderr
<code>_name</code>	(<i>none</i>)	Optional label for diagnostics; otherwise the spawn word’s name

Put your own payload fields directly on the parameter object (for example `.n` or `.pipe`). Keys reserved for the system start with `_`. The whole object is sitting on the child’s data stack when the pip starts.

Pip API

`Pip` is a registered Final type for class members only, such as `newPipParams^`.

Faults

Final panics mean programmer error (stack underflow, bad types, and the like) — not normal control flow. Expected failures should use optional results instead.

A pip must not fail silently. By default, the **Session** is the failure domain: if one pip faults, the whole Session run is treated as faulted. Sibling pips are stopped; there is no “catch in the parent and keep going” path, and still no join. Other Sessions are unaffected.

In practice: close pipes (or otherwise unblock waiters) when shutting down work, so a blocked pull does not hang forever after a fault elsewhere.

Stdio

- By default, `Pip.newPipParams^` copies the current stdout and stderr into the parameter object; the child uses those same writers. Process stdout/stderr are safe for concurrent writes from several pips (including fault reports).

- Custom sinks (builders, files, network, and so on) are **not** automatically locked — your program owns those streams.
- Prefer `fmt` and `write*` on a **TextWrite** / **ByteWrite** for real I/O. Do not treat `stdout` as a general file or socket mux.
- Stdin: line-oriented REPL input stays on the main pip. Worker pips get null stdin unless you bind an explicit **TextRead**.

Pipes

How can a pip safely share information with other pips, or with the pip that spawned it?

The conventional approach at the hardware level, and in many languages that support multi-threading, has been to try to share oftentimes complex data structures or services, by way of locking constructs that enable multiple threads to access things in a way that won't involve the data corruption that occurs when multiple threads read and write multi-value data structures at the same time, clobbering each other's writes, or reading partially modified data that is not correct as a whole. For example, when updating the fields of an address, if a second thread were to read the address while the first thread had only updated the street but not yet the city, the second thread would read an invalid address.

The problem with locks on data structures is that oftentimes several data structures need to be updated for one coherent transaction. This can lead to locking complexity, and "deadlocks" wherein 2 threads manage to obtain one lock of two needed, but not the second. The threads stop. While there have been approaches to this issue that have some limited success, they are fragile, and require precise enforcement. They are also difficult to reason about. Worse, such issues are hard to create under test, and oftentimes only present by way of occasional lockups of processing under load in production.

Pipe Introduction

While Final does provide a few thread-safe data constructs, the predominant way to safely and efficiently communicate between pips is via pipes. ("How many pipes could a pip piper pick if a pip piper could pick pipes?")

A pipe is just what it sounds like: it is a unidirectional conduit which can carry any type of data from a sender to a receiver. It is fully safe on both ends: a sending pip and receiving pip can read/write/check a pipe at the same time, without worry. Typically, one `Pip` has the role of writing to a `pipe`, and another `Pip` has the role of reading from it. For example, the classic case of a producer and consumer: one `Pip` produces some data items, and a second `Pip` consumes them. The act of adding an item to a pipe is called *pushing*; the act of removing an item is called *pulling*.

To work with Pipes, you use the `pipe::` API.

Pipe Capacity and Length

Each `pipe` has a capacity that is set when it is created, and cannot be changed. The capacity determines how many items the pipe can contain before it is full. When a `pipe` is full, a sender trying to push a new item will *block* until at least one item is pulled by a receiver. When a `pipe` is empty, a receiving pip trying to pull an item will block until an item is pushed.

If a `pipe` has zero capacity, then both sides will block until the sender pushes an item and the receiver pulls it. This is typically not too useful, but can be used when you want to synchronize two pips during a multistep operation.

A common case is a `pipe` of length 1. This is often used in a situation to signal the end of an operation, or to communicate an error condition. A receiving pip blocks until the operation is complete, or an error is signalled.

Capacity greater than 1 is used for buffering between a producer and consumer. It can also be used with length of N, for N producer or consumer pips, to determine when all are finished, without needing an additional “done” channel.

To create a pipe, with capacity N:

```
// ( N:i -- p:Pipe )
N pipe::new
```

To query the number of items currently in the Pipe, use the `pipe::len` operator. NOTE: this value can change after reading in multi-threading uses due to pushes or pops made by other threads. The examples show a scenario where using `pipe::len` is safe and useful.

```
// ( p:Pipe -- len:i )
myPipe pipe::len
```

Pushing/Popping Data into/out of a Pipe

To push an item into a Pipe, use `>pipe`. This operation will block if the Pipe is full.

To pull an item from a Pipe, use `pipe>`. This operation will block if the Pipe is empty.

To try to pull an item from a Pipe, without blocking, use `pipe>?`. This operation returns a value and true if available, or false if no value is available. Do not use this to

create a tight polling loop, which doesn't release the thread for other uses; use it only when you can alternatively do or test something else being available to do work.

```
// ( item:a p1:Pipe -- )
true myPipe >pipe // eg, push a true boolean into the pipe

// ( p2:Pipe -- item:a )
myPipe pipe> // blocks until the item is available

// ( p2:Pipe -- item:a true | false )
myPipe pipe>? // don't block, if item not available
```

Iterating a Pipe

A particularly convenient operation on a receiving pipe that will receive a possibly unknown number of requests from a sender, who will then close the pipe when finished, is to iterate the pipe using `foreach`. This operation will return a new value `i` from the pipe when available; the loop will exit when the sender closes the pipe.

To iterate a receiving Pipe that the sender will close when done:

```
// ( p:Pipe -- )
myReceivingPipe
foreach // blocks until a value is available or pipe is closed
  i // handle the value popped from the pipe
/foreach
// the pipe has been closed, no more data will be sent
```

Closing a Pipe

A sender can optionally `close` a pipe. This lets the receiver know that no more values will be sent. When using a pipe that will be closed by the sender, the receiver should use a `foreach` on the pipe, which will return each value `i` when it is received, and stop iterating when the `pipe` is closed. This mode of using a pipe is optional; `close` does not release resources, and should only be called for use-cases involving receiver iteration. A receiver should NEVER close a sender's pipe, as this will cause a fault if the sender tries to push more data.

To close a Pipe:

```
// ( p:Pipe -- )
mySendingPipe pipe::close // nop if already closed
```


To check if a Pipe is closed. NOTE: this is a generally unsafe api, since the pipe can become closed by a sender between the time a receiver makes this call, and then attempts to read. Use `foreach` iteration on a receiving pipe that will be closed when no longer needed.

```
// ( p:Pipe -- closed:b )  
mySendingPipe pipe::closed?
```

Ticker and Timer Pipes

Final provides prebuilt **modal** Pipes that push `DateTime` either periodically (ticker) or once after an interval / at a clock time (timer).

Factories create an **unarmed** pipe. Lifecycle words both **initially arm** and **renew**.

Ticker Pipe

```
Pipe.newTicker^ l.tick!
Interval.second l.tick@ pipe::tickerStart
// ...
l.tick@ pipe::tickerStop
```

- `pipe::tickerStart ( iv:Interval p:Pipe -- )` — start or restart
- `pipe::tickerStop ( p:Pipe -- )` — stop ticks; pipe stays open
- Do **not** `pipe::close` a ticker pipe (fault)
- Do `pipe::tickerStop` when no more ticks are needed

Timer Pipe

```
Pipe.newTimer^ l.boom!
500 Interval.millisecond dt:* l.boom@ pipe::timerAfter
// or: someDateTime l.boom@ pipe::timerWhen
```

- `pipe::timerAfter ( iv:Interval p:Pipe -- )` — arm or re-arm after interval
- `pipe::timerWhen ( when:DateTime p:Pipe -- )` — arm or re-arm at time (past → immediate)
- Does **not** self-close after fire; idle if not rearmed is harmless
- Do **not** `pipe::close` a timer pipe (fault)

Example

See `examples/threading/tick-tick-boom.fnl` for both types of Pipes. Note that the example only uses the Timer pipe once, but you can easily use it in a re-fire mode, by re-arming it after it signals.

Atomics

Shared state (optional; prefer pipes)

The preferred method of ITC is via `pipes` (“share by communicating”).

However, thread-safe variables and collections can have their use, so Final provides a curated collection of thread-safe constructs.

AtomicInt

A single thread-safe integer. Readonly `.value`; mutating methods return the **new** value except `.set^` / `.zero^`.

Kind	Name	Effect	Description
class	<code>new</code>	<code>(-- a:AtomicInt)</code>	starts at 0
prop	<code>value</code>	<code>Int</code>	current value (const)
method	<code>inc</code>	<code>(-- n:i)</code>	add 1; return new value
method	<code>dec</code>	<code>(-- n:i)</code>	subtract 1; return new value
method	<code>add</code>	<code>(n:i -- n':i)</code>	add <code>n</code> ; return new value
method	<code>sub</code>	<code>(n:i -- n':i)</code>	subtract <code>n</code> ; return new value
method	<code>set</code>	<code>(n:i --)</code>	store <code>n</code>
method	<code>zero</code>	<code>(--)</code>	store 0

```
AtomicInt.new^ >ts
t:.inc^ $ nl      // 1
10 t:.add^ $ nl   // 11
t:.value@ $ nl    // 11
t:drop
```

Prefer a Pipe when you are passing work between Pips. Use `AtomicInt` when several Pips only need to bump a counter.

AtomicList / AtomicStringList

These are mutex-guarded Lists for reading/writing an item.

Iteration via `foreach` uses a snapshot clone, so is threadsafe. NOTE: do NOT iterate `AtomicLists` with a range-based iterator or other counted loop; iteration is not threadsafe using that approach.

AtomicDynamicObject

A mutex-guarded `DynamicObject` (string keys) that is threadsafe for reading a key, writing a key value, and adding a new field. `foreach` uses a snapshot clone, similar to the `AtomicLists`.

Average and MovingAverage

`Average` is a type used for accumulating an exactly correct mean and variance for data samples added via the `add` method, without retaining the samples. `MovingAverage` is used for calculating a recent, moving average. Both are threadsafe for `add`, and the atomic read methods.

See `help Average` and `help MovingAverage` for the apis.

See [Counter / Average / MovingAverage](#)

atomicVar:

A mutex guarded var, threadsafe for fetch and set (`@ / !`).

Not that `const:` is inherently threadsafe, as long as the definition precedes any use.

Note that both `atomicVar:` and `const:` write to the dictionary, so are disallowed when Pips are running. And both run in `interpret/immediate` mode, so can not even be invoked by Pips.

General Caveats

In all the collections above, threadsafety extends only to the operations listed. In particular, the threadsafety of any objects stored in these containers have the threadsafety of the object itself, which for most Objects is NOT threadsafe.

Threadsafe Operations

- reading any class-level sealed readonly/const property; eg `Math.pi`
- reading properties from object instances that are not mutated over the lifetime of the reading Pips

- creating/mutating objects created by a Pip, and not shared
- reading/mutating an object created by a Pip, then sent via a Pipe to another Pip, under contract that the sending Pip then no longer mutates or uses the object; eg. preparing a `DynamicObject` by `Pip1`, then sending it to `Pip2`, abandoning its ownership
- calling object factories, such as opening new file streams
- writing to the shared `stdout/stderr`, eg using `$/$$`

Web/Http

Final's HTTP support covers **client-side dialing** and **server-side listening**:

- **Dialing** — build an `HttpRequest`, send it with an `HttpDialer` (or the `httpDial` helper), and read an `HttpRes`. Everyday HTTPS GET/POST against remote hosts.
- **Listening** — bind an `HttpListener`, register ServeMux routes, spawn Pip handlers per route, and serve inbound requests.

You already know the pieces these APIs lean on:

- **Class methods** with `^`, property `@ / !`, and nested paths like `t!.hostAddress.host!` — [Methods, Compound Path Syntax](#)
- Parking a request or dialer on `ts` for a short session of member ops — [Writing Clean Final Code](#)
- **StringKeyValueList** for dial query parameters and headers (duplicate keys allowed) — [StringKeyValueList](#)
- The **modal result** shape `( ... -- result true \ | err:s false )` — [Writing Clean Final Code](#)
- Parsing a JSON response body with `json>` — [Json](#). To *send* JSON, `setBodyAsJsonString` on `HttpRequest` / `HttpOutRes` takes an already-rendered JSON string and sets `Content-Type`. Request and response bodies can also be **TextRead** / **ByteRead** streams — [Dialing, Streams](#).
- **Pips** and **Pipes** for listen-side concurrency — [Threading](#)

Page	Contents
Dialing (Get/Post)	<code>HttpAddress</code> , <code>HttpRequest</code> , <code>HttpDialer</code> , <code>HttpRes</code> , helpers, worked examples
Listening (Serving)	<code>HttpListener</code> , <code>HttpServeJob</code> , ServeMux routes, path params, Pip-per-route handlers

Use `help HttpRequest` / `help HttpListener` (and the other type names) in the TUI for the full member list and stack effects.

Dialing (Get/Post)

Dialing means making an outbound HTTP request and reading the response. The flow is always the same:

1. Create an `HttpReq` and set at least `hostAddress.host`.
2. Optionally set path, query, headers, method, timeout, and body.
3. Send with `HttpDialer.send` (or `httpDial`).
4. On transport success, inspect `HttpRes.status`, headers, and body.

Transport failures (DNS, timeout, TLS, ...) use the modal shape `( -- err:s false )`. A completed HTTP exchange — including **4xx/5xx** — returns `( -- res:HttpRes true )`; check `.status` yourself.

Objects

HttpAddress

Target of the request: scheme, host, and optional port. Every `HttpReq` owns one under `.hostAddress` (read-only property — mutate its fields).

Kind	Name	Type / Effect	Description
class	<code>new</code>	<code>(-- addr)</code>	empty address
prop	<code>scheme</code>	String	"http" or "https"; empty → https when dialing
prop	<code>host</code>	String	host name or IP; required before send
prop	<code>port</code>	Int	<code>0</code> → scheme default (80 / 443)

Typical setup parks the request on `ts` and writes the nested host:

```
HttpReq.new^ >ts
"checkip.amazonaws.com" t:.hostAddress.host!
ts>
```

HttpReq

Outgoing request. Only `hostAddress.host` is required; everything else has defaults.

Kind	Name	Type / Effect	Description
class	<code>new</code>	<code>(-- req)</code>	fresh req with empty address, query list, and headers
prop	<code>timeoutS</code>	<code>Int</code>	seconds; <code>0</code> → no explicit timeout (see Notes)
prop	<code>hostAddress</code>	<code>HttpAddress</code>	owned address (mutate fields; property itself is read-only)
prop	<code>method</code>	<code>String</code>	e.g. "GET" , "POST" ; empty → POST if body set, else GET
prop	<code>path</code>	<code>String</code>	URL path; leading <code>/</code> optional
prop	<code>queryParams</code>	<code>StringKeyValueList</code>	ordered query pairs
prop	<code>headers</code>	<code>StringKeyValueList</code>	ordered request headers
method	<code>setBodyFromString</code>	<code>(s:String --)</code>	set body from UTF-8 text (does not set Content-Type)
method	<code>setBodyFromBytes</code>	<code>(bytes:ByteList -)</code>	set body from Bytes (<code>ByteList</code> carrier; no Content-Type default)
method	<code>setBodyFromTextRead</code>	<code>(r:TextRead --)</code>	set body from a text stream (does not set Content-Type)
method	<code>setBodyFromByteRead</code>	<code>(r:ByteRead --)</code>	set body from a byte stream (no

Kind	Name	Type / Effect	Description
			Content-Type default)
method	<code>setBodyAsJsonString</code>	<code>(json:s --)</code>	set body from an already-rendered JSON string and Content-Type: application/json
method	<code>clearBody</code>	<code>(--)</code>	clear body (reuse req for a later GET)
method	<code>appendPathFromList</code>	<code>(segments:List</code>	join String/Int/Uin segments onto

Query and headers use `!{}` (upsert first key) or `.add^` (always append a duplicate). See [StringKeyValueList](#).

```
HttpReq.new^ >ts
"wttr.in" t:.hostAddress.host!
"Denver" t:.path!
"3" "format" t:.queryParams!{}
"text/plain" "Accept" t:.headers!{}
10 t:.timeoutS!
ts>
```

HttpDialer

Stateless sender. Create one, call `send`, then drop it — or use `httpDial`, which does that for you.

Kind	Name	Effect	Description
class	<code>new</code>	<code>(-- dialer)</code>	new dialer
method	<code>send</code>	<code>(req:HttpReq -- res:HttpRes true err:s false)</code>	prepare + transport

```
HttpDialer.new^ >ts
t:.send^          // consumes req from DS
t:drop
```

HttpRes

Result of a successful transport. There is no public `new`; only `send` produces one.

Kind	Name	Type / Effect	Description
prop	<code>status</code>	<code>Int</code>	HTTP status, e.g. 200, 404
prop	<code>headers</code>	<code>StringKeyValueList</code>	one pair per header line (duplicates kept)
method	<code>getBodyAsString</code>	<code>(-- s:String)</code>	body as UTF-8 text
method	<code>getBodyAsBytes</code>	<code>(-- bytes:ByteList)</code>	body as Bytes (<code>ByteList</code> carrier)
method	<code>getBodyAsTextRead</code>	<code>(-- r:TextRead)</code>	body as a text stream
method	<code>getBodyAsByteRead</code>	<code>(-- r:ByteRead)</code>	body as a byte stream

Helpers

Core ships two convenience words (see `0068-http_util.fnl`):

Word	Effect	Description
<code>httpDial</code>	<code>(req:HttpReq -- res:HttpRes true err:s false)</code>	<code>HttpDialer.new^ + .send^</code>
<code>printHttpRes</code>	<code>(res:HttpRes true err:s false --)</code>	print status+body, or ERROR: ...

`printHttpRes` consumes the modal pair left by `httpDial` / `.send^`.

Examples

These follow the same progression as `examples/http/dial/http-dial_0-simple.fnl` and `http-dial_1-advanced.fnl`.

Simple GET (host only)

```
: makeCheckIpReq
  HttpRequest.new^
>ts
"checkip.amazonaws.com" t:.hostAddress.host!
10 t:.timeoutS!
ts>
;

makeCheckIpReq httpDial printHttpRes
```

GET with a path

```
: makeZenReq
  HttpRequest.new^
>ts
10 t:.timeoutS!
"api.github.com" t:.hostAddress.host!
"zen" t:.path!
ts>
;

makeZenReq httpDial printHttpRes
```

Path, query, and Accept header

```
: makeWttrReq
  HttpRequest.new^
>ts
"wttr.in" t:.hostAddress.host!
"Denver" t:.path!
"3" "format" t:.queryParams!{}
"text/plain" "Accept" t:.headers!{}
10 t:.timeoutS!
ts>
;

makeWttrReq httpDial printHttpRes
```

JSON response body

`HttpRes` returns the body as text, bytes, or a stream. Parse JSON yourself with `json>` ([Json Parsing](#)):

```

: makeBingoJsonRes
  HttpReq.new^
  >ts
  "httpbingo.org" t:.hostAddress.host!
  "anything/hello" t:.path!
  "bar" "foo" t:.queryParams!{}
  10 t:.timeoutS!
  ts>
;

makeBingoJsonRes httpDial
dup2 printHttpRes
if
  >ts t:.getBodyAsString^ json> t:drop
  // DynamicObject / List on DS – walk as usual
else
  drop          // discard err string; already printed
then

```

POST JSON

Empty `.method` becomes **POST** once a body is set. Set **Content-Type** yourself:

```

HttpReq.new^ >ts
"httpbingo.org" t:.hostAddress.host!
"post" t:.path!
"application/json" "Content-Type" t:.headers!{}
`{"hello":"world","n":42}` t:.setBodyFromString^
10 t:.timeoutS!
ts>
httpDial printHttpRes

```

To reuse the same req for a later GET, call **clearBody** so the method default returns to GET.

Bodies as streams

You can feed the request from a **TextRead** / **ByteRead**, and read the response the same way. Useful when the payload is already a stream, or when you want to pull the response incrementally ([Streams](#)).

POST text from a String reader:

```

HttpReq.new^ >ts
"httpbingo.org" t:.hostAddress.host!
"post" t:.path!
"POST" t:.method!
"text/plain" "Content-Type" t:.headers!{}
"hello stream" TextReader.newFromString^ drop
t:.setBodyFromTextRead^
15 t:.timeoutS!
ts>
httpDial printHttpRes

```

Read a response as text lines:

```

HttpReq.new^ >ts
"httpbingo.org" t:.hostAddress.host!
"stream/3" t:.path!
15 t:.timeoutS!
ts>
httpDial
if
  >ts
  t:.getBodyAsTextRead^ >xs
  x:.readLine^ if $ nl then
  x:.close^ x:drop
  t:drop
else
  drop
then

```

For binary bodies (images, opaque octets), use `setBodyFromByteRead` / `getBodyAsByteRead` with a **ByteList** reader — see `examples/http/dial/http-dial-demo.fnl` (`dialStreamReqByte` , `dialStreamResByte`).

Transport error

A bad host yields the false branch (timeout / DNS), not an `HttpRes` :

```

HttpReq.new^ >ts
4 t:.timeoutS!
"badbadbadbad123456.com" t:.hostAddress.host!
ts>
httpDial printHttpRes

```

Notes

- **Status vs transport** — `true` means the HTTP exchange finished. Always check `.status` for application success.
- **Timeout** — if `.timeoutS` is `0` at send time, the dialer may apply the session's advisory run budget; set an explicit seconds value for predictable client timeouts.
- **No cookie jar** — each send is independent; cookies are not persisted across calls.
- **Security** — `HttpReq`, `HttpDialer`, and `HttpRes` participate in the **HttpDial** restriction. Restricted embeds / AI hosts may deny dialing; see [Security Restrictions](#).
- **Examples on disk** — `examples/http/dial/http-dial-demo.fnl`, `http-dial_0-simple.fnl`, `http-dial_1-advanced.fnl`.

Listening (Serving)

Listening means binding an HTTP server and handling inbound requests. Final's model is:

1. Create an `HttpListener` and set `bind address` (host + port).
2. For each route: allocate a **request Pipe**, **spawn** a handler Pip that loops on that Pipe, then **register** the ServeMux pattern with `.handle^` (or `.get^` / `.post^` / ...).
3. **spawn**: a server Pip that calls `.listen^` — never block the main / REPL pip on listen.
4. Per request, the runtime pushes an `HttpServeJob` on the route's request Pipe; the handler writes `.res`, then signals `.reply`.

Routing uses Go's `http.ServeMux` patterns (`METHOD /path/{name}`). Matching, specificity, and default 404/405 follow ServeMux — Final does not implement its own matcher. Path wildcards land on `HttpInReq.pathParams` (e.g. `GET /api/items/{id}` → `x:req.pathParams.id@`). Query keys land on `HttpInReq.queryParams` (`?id=1`). They are not interchangeable: `/api/items/1` matches a `{id}` route; `/api/items/?id=1` does not turn into a path param.

Objects

HttpListener

Stateful server. You must `new^` — listening is not a one-shot helper.

Kind	Name	Type / Effect	Description
class	<code>new</code>	<code>(-- listener)</code>	empty listener; owns ServeMux + route table
prop	<code>address</code>	<code>HttpAddress</code>	bind address (mutate <code>.scheme</code> , <code>.host</code> , <code>.port</code>)
prop	<code>routes</code>	<code>List</code>	registered <code>HttpRoute</code>

Kind	Name	Type / Effect	Description
			entries (read-only)
prop	<code>readTimeoutS</code>	Int	0 → no extra read timeout
prop	<code>writeTimeoutS</code>	Int	0 → no extra write timeout
method	<code>handle</code>	<code>(pattern:s name:s xt:XT requestPipe:Pipe --)</code>	full ServeMux pattern, e.g. "GET /health"
method	<code>get</code>	<code>(path:s name:s xt:XT requestPipe:Pipe --)</code>	same as <code>handle</code> with "GET " prepended
method	<code>post</code>	<code>(path:s name:s xt:XT requestPipe:Pipe --)</code>	"POST " + path
method	<code>put</code>	<code>(path:s name:s xt:XT requestPipe:Pipe --)</code>	"PUT " + path
method	<code>delete</code>	<code>(path:s name:s xt:XT requestPipe:Pipe --)</code>	"DELETE " + path
method	<code>listen</code>	<code>`(- true</code>	<code>err:s false)`</code>
method	<code>shutdown</code>	<code>`(- true</code>	<code>err:s false)`</code>

`address.port` is **required** before `listen`. Empty host defaults to `127.0.0.1`. For other methods (e.g. PATCH), pass a full pattern to `.handle^`.

HttpServeJob

Per-request job object pushed on the route's request Pipe. There is no public `new`; only the listener creates jobs.

Kind	Name	Type	Description
prop	<code>req</code>	HttpInReq	inbound request
prop	<code>res</code>	HttpOutRes	outbound response (write this)
prop	<code>reply</code>	Pipe	per-request ; push a done token when finished

Always signal `.reply` (including error paths), or the Go side hangs waiting for the response flush.

HttpInReq

Read-focused inbound request (no public `new`).

Kind	Name	Type / Effect	Description
prop	<code>method</code>	String	e.g. "GET"
prop	<code>path</code>	String	path only (no query), leading /
prop	<code>pathParams</code>	DynamicObject	one String per <code>{name}</code> from the pattern
prop	<code>queryParams</code>	DynamicObject	one value per key (no multi-value)
prop	<code>headers</code>	StringKeyValueList	request headers
prop	<code>host</code>	String	Host header / host
prop	<code>remoteAddr</code>	String	peer address
method	<code>getBodyAsString</code>	<code>(-- s)</code>	body as UTF-8 text
method	<code>getBodyAsBytes</code>	<code>(-- bytes:ByteList)</code>	body as Bytes
method	<code>getFormParams</code>	<code>`(- form:StringKeyValueList true</code>	<code>err:s false)`</code>

HttpOutRes

Writable outbound response. Flushed after the reply Pipe signals. Status `0` → **200** at flush. Status is **not** a header.

Kind	Name	Type / Effect	Description
prop	<code>status</code>	Int	HTTP status line; unset/ <code>0</code> → 200
prop	<code>headers</code>	StringKeyValueList	response headers
method	<code>setBodyFromString</code>	<code>(s:String --)</code>	set body (does not set Content-Type)
method	<code>setBodyFromBytes</code>	<code>(bytes:ByteList --)</code>	set body from Bytes
method	<code>setBodyAsJsonString</code>	<code>(json:s --)</code>	JSON string body + Content-Type: application/json
method	<code>clearBody</code>	<code>(--)</code>	clear buffered body

Concurrency model (Pip-per-route)

Native ServeHTTP **never** spawns Final pips. You spawn handlers in Final **before** listen:

```

setup pip
  → spawn handler loop on requestPipe[i]
  → register pattern → stub closed over i
  → spawn: serverPip → listener.listen^

per HTTP request (Go stub):
  → new reply Pipe
  → push HttpServeJob(req, res, reply) on requestPipe[i]
  → wait on reply → flush .res → return

handler pip:
  → pipe> job → write .res → push done on .reply

```

One serial loop Pip per route. Give the request Pipe capacity (e.g. `Pipe.newWithCapacity^`) so bursts can queue while a handler is busy.

Server pip: `.listen^` blocks **that** pip. Main continues; call `.shutdown^` from another pip to stop.

ServeMux patterns and REST params

Pattern		Notes
GET /health		method + exact path
GET /api/items/{id}		path param → <code>x::req.pathParams.id@</code>
POST /api/items		method-specific
PUT /api/items/{id}		via <code>.put^</code> or <code>.handle^</code>

Source	Access
Path {id}	<code>x::req.pathParams</code> <code>DynamicObject</code> (Strings; coerce with <code>s>i</code> etc.)
Query	<code>x::req.queryParams</code> <code>DynamicObject</code>
Form urlencoded	<code>x::req.getFormParams^</code>
Headers	<code>x::req.headers</code> / <code>x::res.headers</code> <code>StringKeyValueList</code>
Body	<code>getBodyAsString</code> / <code>getBodyAsBytes</code> ; parse JSON with <code>json></code> if needed

Example

Matches `examples/http/listen/http-listen_0-simple.fnl` (health + path-param GET).

```

20 Pipe.newWithCapacity^ const: healthReqs
20 Pipe.newWithCapacity^ const: getItemReqs

: healthLoop ( pp:DynamicObject -- )
  >zs
  z:.requestPipe@ >ys
  forever
    y:peek pipe> >xs          // HttpServeJob
    200 x:.res.status!
    "text/plain" "Content-Type" x:.res.headers!{}
    "ok" x:.res.setBodyFromString^
    0 x:.reply@ >pipe          // done token → flush
    x:drop
  /forever
;

: getItemLoop ( pp:DynamicObject -- )
  >zs
  z:.requestPipe@ >ys
  forever
    y:peek pipe> >xs
    200 x:.res.status!
    "text/plain" "Content-Type" x:.res.headers!{}
    "id was: " x:.req.pathParams.id@ str::+ x:.res.setBodyFromString^
    0 x:.reply@ >pipe
    x:drop
  /forever
;

HttpListener.new^ const: myListener

: bootHandlers ( -- )
  Pip.newPipParams^ >ts
  healthReqs t:.requestPipe!
  ts> spawn: healthLoop
  Pip.newPipParams^ >ts
  getItemReqs t:.requestPipe!
  ts> spawn: getItemLoop
;

: registerRoutes ( -- )
  myListener >ts
  "GET /health"          "healthLoop" "healthLoop" find .xt nip
healthReqs t:.handle^
  "GET /api/items/{id}" "getItemLoop" "getItemLoop" find .xt nip
getItemReqs t:.handle^
  t:drop
;

: runServer ( pp -- )

```

```

drop
myListener >ts
"http" t:.address.scheme!
"127.0.0.1" t:.address.host!
18080 t:.address.port!
t:.listen^
if
  "listener stopped\n" $
else
  "listen ERROR: %s\n" fmt nip $
then
  t:drop
;

: spawnServer ( -- )
  Pip.newPipParams^ spawn: runServer
;

bootHandlers
registerRoutes
spawnServer

```

Then: `GET http://127.0.0.1:18080/health` and `GET http://127.0.0.1:18080/api/items/42`.

Notes

- **Spawn before listen** — handler pips must be running before `.listen^`.
- **Reply Pipe** — capacity-1 Pipe in the job; push any done token (examples use `0`).
- **Status vs transport** — handler owns `.res.status` (4xx/5xx). Reply means “flush,” not success.
- **Security** — listener types participate in `SR.HttpListen`. Spawning server and handler pips needs `SR.Threads`. See [Security Restrictions](#).
- **Dialing** — outbound client APIs are separate; see [Dialing \(Get/Post\)](#).
- **Examples on disk** — `examples/http/listen/http-listen_0-simple.fnl`.

Use `help HttpListener` (and `HttpInReq`, `HttpOutRes`, `HttpServeJob`) in the TUI for the full member list.

SQL Databases

Final's SQL API is **provider-independent**: one type, `DbProvider`, wraps Go `database/sql`. A class constant names the driver; the same methods work for every engine.

Supported drivers:

Constant	Engine	Notes
<code>DbProvider.postgres</code>	Postgres / AWS RDS (pgx)	Placeholders <code>\$1</code> , <code>\$2</code> , ...
<code>DbProvider.mysql</code>	MySQL / MariaDB	Placeholders <code>?</code> ; prefer <code>parseTime=true</code> in the DSN
<code>DbProvider.sqlite</code>	SQLite (modern, CGo-free)	DSN is a path or <code>":memory:"</code>

There is no ORM and no query builder — you write SQL, bind ordinal args from a `List`, and consume rows.

The `final` and `final-go` binaries include all three drivers. `finalgo` / `finalembded` do not import SQL drivers; the embedding host declares what it needs. The Pi pack includes SQLite.

Opening a connection is gated by `SR.Db`. SQLite also needs `SR.FileRead` / `SR.FileWrite` because it opens a file.

Connection

```
DbProvider.postgres dsn DbProvider.new^
// ( driver:s dsn:s|Moniker -- db:DbProvider true | err:s false )
```

Kind	Name	Effect	Description
class prop	<code>postgres / mysql / sqlite</code>	<code>(-- driver:s)</code>	driver name for <code>new</code>
class	<code>new</code>	<code>(driver:s dsn:s Moniker -- db true</code>	<code>sql.Open + Ping</code>

Kind	Name	Effect	Description
		err:s false)	

The DSN may be a **String** or a **Moniker** (opaque handle — see [Env and Moniker](#)). Keep real passwords out of committed source.

Postgres with Moniker

```
`postgres://$POSTGRES_UNAME@localhost:5432/final_demo?
sslmode=disable&password=$POSTGRES_PWD`
const: hpDsn

: createDsnMoniker
  hpDsn Moniker.new^ >ts
  t:peek "POSTGRES_UNAME" Env.subMoniker^ >xs
  t:peek "POSTGRES_PWD" Env.subMoniker^
  xs> && if
    ts>
  else
    t:drop
    "Could not substitute required Postgres parameters" error
  then
;

: connectDb
  DbProvider.postgres createDsnMoniker
  DbProvider.new^
  nif
    dup "Can't open DbProvider: " $ $ nl
    error
  then
;
```

MySQL / SQLite sketches

```
DbProvider.mysql
"user:pass@tcp(127.0.0.1:3306)/final_demo?parseTime=true"
DbProvider.new^

DbProvider.sqlite ":memory:" DbProvider.new^
DbProvider.sqlite "/tmp/demo.db" DbProvider.new^
```

Exec and query

Kind	Name	Effect	Description
method	<code>exec</code>	<code>(sql:s --)</code>	DDL / DML, no binds; panics on error
method	<code>execWithArgs</code>	<code>(sql:s args:List --)</code>	ordinal binds; panics on error
method	<code>query</code>	<code>(sql:s -- iter:DbIter)</code>	result iterator
method	<code>queryWithArgs</code>	<code>(sql:s args:List -- iter:DbIter)</code>	query with binds

Args are a **List in ordinal order** — build with `( )`. Use a **ByteList** for BLOB columns. Placeholder syntax follows the driver (`$N` vs `?`); always pass values by position, not by name.

```
: insertCharList  // ( args:List -- )
,
INSERT INTO "hpCharacters" ("firstName", "lastName", "isGood",
"firstMovieSeenYear")
VALUES ($1, $2, $3, $4)
,
  hpDb @ >ts
    swap t:.execWithArgs^
  t:drop
;

( "Harry" "Potter" true 2001 ) insertCharList
```

Result rows

`.query^` / `.queryWithArgs^` return a **DbIter**: foreach-compatible. Each loop value `i` is a **StaticObject** whose fields are the result-set column names (creation order matches the SELECT), minted from the iterator's shared prototype (`iter.protoObj`). Cell types follow the driver's Scan values, coerced to Final (`Int`, `String`, `Bool`, `Float`, `ByteList`, `DateTime` where applicable, ...). SQL NULL is Final **Null**.

Row objects share one schema prototype DO built from the column list. For fixed-schema in-memory grids, see [StaticObject](#) and [Table](#).


```

: queryAll
  hpDb @ >ts
,
SELECT "id", "firstName", "lastName", "isGood", "firstMovieSeenYear"
FROM "hpCharacters"
ORDER BY "firstMovieSeenYear", "lastName", "firstName"
,
  t:.query^
  t:drop

foreach
  i >ts
    ( t:.firstName@ t:.lastName@ t:.firstMovieSeenYear@ )
    "name: %s %s  year first seen: %d\n" fmt $
  t:drop
/foreach
;

```

Column names and anonymous queries

Unaliased expressions get **provider-supplied** names. Inspect them with:

```
iter .getColumnNames^  // ( -- names:StringList )
```

Prefer explicit aliases in SQL when you want stable Final field names:

```
SELECT count(*) AS "n", sum("firstMovieSeenYear") AS "yearSum" ...
```

NULL helpers

Word	Effect	Description
val?	(val -- val true false)	false if Null
val def	(val default -- val default)	substitute default for Null

Materialize with iter>list and dbIter>table

These are core utility words (not something you have to define):

```

iter>list      // ( iterable -- l:List )
dbIter>table   // ( iter:DbIter -- tbl:Table )

```

`iter>list` drains any `foreach`-compatible source into a **List**. For a query, that is a List of row **StaticObjects**.

```
sql hpDb @ .query^ iter>list >jsonp
```

`dbIter>table` builds a **Table** from a query iterator: schema from the iterator's `DynamicObject` prototype, rows as `StaticObjects`. See [Table](#).

```
sql hpDb @ .query^ dbIter>table >ts
t:.rows >jsonp $
t:drop
```

Notes

- `DbProvider.new^` is **modal**; `exec` / `query` **panic** if the SQL cannot run.
- Empty result: `foreach` runs zero times; `iter>list` yields an empty `List`; `dbIter>table` yields a `Table` with schema and no rows.
- Full walkthrough: `examples/db/postgres/harry-potter.fnl`.
- Discoverability: `help DbProvider` in the TUI.

Related: [Env and Moniker](#).

Env and Moniker

Env reads host configuration. **Moniker** holds a string template that may contain `$KEYNAME` placeholders, filled without putting secrets on the data stack. Together they let scripts connect to databases (and similar services) without baking credentials into `.fnl` source.

Env

Class-only type — no instances. Lookup order:

1. `./.env` in the process **current working directory** (re-read on every call; not cached)
2. **Process environment** (`os.LookupEnv`)

A key in `.env` wins over the same key in the process environment. Keys are **case-sensitive**.

Kind	Name	Effect	Description
class	<code>get</code>	<code>(key:s -- val:s true false)</code>	cleartext value on DS when found
class	<code>subMoniker</code>	<code>(mon:Moniker keyname:s -- b)</code>	replace <code>\$keyname</code> in moniker; never pushes the secret

```
"POSTGRES_HOST" Env.get^
if
  // val:s on stack
  $ nl
else
  "POSTGRES_HOST not set" $ nl
then
```

Empty string is a **found** value (`"" true`), distinct from missing (`false`).

`.env` grammar (cwd)

Line	Meaning
blank	ignored
<code># ... or // ...</code>	comment

Line	Meaning
<code>export VAR=value</code>	same as <code>VAR=value</code>
<code>VAR=value</code>	binding (split on first <code>=</code> ; trim key and value)

Double-quoted values keep interior/trailing spaces after the quotes are removed. No `${...}` expansion, no multiline values. Duplicate keys: earlier binding wins. Missing `.env` is normal — lookup falls through to the process environment.

Moniker

Opaque string handle. Create from a clear template; fill placeholders with `Env.subMoniker^`. Final never prints the payload — diagnostics show `Moniker(<opaque>)`.

Kind	Name	Effect	Description
class	<code>new</code>	<code>(init:s -- mon:Moniker)</code>	template or plain string

Placeholder form: literal `$` + key name (e.g. `$POSTGRES_PWD`). No `${KEY}` form. Call `subMoniker` **once per key**. Unresolved `$...` tokens remain until a source returns `true`.

```
"postgres://user:$POSTGRES_PWD@localhost/db" Moniker.new^ >ts
t:peek "POSTGRES_PWD" Env.subMoniker^
if
  // substituted; moniker ready for DbProvider.new^
else
  // try another source, or error
then
```

`Env.subMoniker^` returns `false` when the key is missing and leaves the moniker unchanged — suitable for a waterfall of sources.

Use as DSN

`DbProvider.new^` accepts **String or Moniker** for the DSN. Trusted Go code unwraps the moniker internally; the resolved string never lands on the Final data stack.

```

`postgres://$POSTGRES_UNAME@localhost:5432/final_demo?
sslmode=disable&password=$POSTGRES_PWD`
const: hpDsn

: createDsnMoniker
  hpDsn Moniker.new^ >ts
  t:peek "POSTGRES_UNAME" Env.subMoniker^ >xs
  t:peek "POSTGRES_PWD" Env.subMoniker^
  xs> && if
    ts>
  else
    t:drop
    "Could not substitute required Postgres parameters" error
  then
;

DbProvider.postgres createDsnMoniker DbProvider.new^

```

Prefer this path over `Env.get^` when the value must stay out of scripts, logs, and `$` / JSON dumps.

When to use which

Need	API
Read a non-secret config string in Final	<code>Env.get^</code>
Build a DSN / URL that embeds secrets	<code>Moniker.new^ + Env.subMoniker^</code>
Pass credentials into <code>DbProvider</code>	Moniker (or String if you accept cleartext on DS)

Non-goals

Env does not write the process environment, walk parent directories for `.env`, or interpolate variables inside `.env` values. Moniker does not expose accessors that return the current string to Final.

See also: [SQL Databases](#).

Cryptography

Final exposes a **small, modern crypto API**: digests, AEAD, signing, key agreement, MACs, KDFs, and a CSPRNG. Algorithms are selected like database drivers — **class constants** on one type per kind (`Hash.sha256` , `Cipher.aes256gcm` , ...), not a plugin layer and not one Final type per algorithm.

Type	Role
<code>Hashes</code>	Unkeyed digests (MD5 checksum, SHA-256)
<code>Hex / Base64</code>	Encode digests and key bytes for text
<code>SecureRandom</code>	Cryptographic random bytes
<code>Keys</code>	Symmetric and asymmetric key material
<code>Ciphers</code>	AEAD encrypt / decrypt
<code>Mac</code>	HMAC-SHA-256
<code>Kdf</code>	HKDF-SHA-256, Argon2id

Policy: recommended algorithms only (no DES, RC4, RSA-PKCS1v1.5, ...). APIs are **one-shot** (whole payload → result). Incremental `update` / `digest` is not available.

Privileged members require `SR.Crypto` . Examples live under `examples/crypto/` .

Pattern

```
bytes Hash.sha256 Hash.hash^ Hex.encode^
Key.aes256gcm Key.generate^
Cipher.aes256gcm swap Cipher.new^
```

Multi-argument, algorithm-dependent inputs use a **params DynamicObject** (prefer `{ value name ... }` for ad-hoc DynamicObjects).

Hashes

Class-only `Hash`. One-shot digests.

Constant	Use
<code>Hash.md5</code>	Non-cryptographic checksum — cheap object / cache keys only
<code>Hash.sha256</code>	Default content-id / integrity hash

```
// ( bytes:ByteList|String algo:s -- digest:ByteList )
"abc" Hash.sha256 Hash.hash^ Hex.encode^    // ba7816bf8f01cfea...
blob Hash.md5 Hash.hash^                    // 16-byte checksum
```

Input: `ByteList` preferred, or `String` as UTF-8 bytes. Output: `ByteList` digest; use [Hex](#) / [Base64](#) for text.

Hex and Base64

Class-only helpers for digests, signatures, and key bytes on text channels. Require `SR.Crypto`.

Hex

Method	Effect
<code>Hex.encode</code>	<code>(bytes:ByteList String -- hex:s)</code> lowercase hex
<code>Hex.decode</code>	<code>(hex:s -- bytes:ByteList)</code> panics on invalid hex

Base64

Standard encoding (not raw/URL variant).

Method	Effect
<code>Base64.encode</code>	<code>(bytes:ByteList String -- s)</code>
<code>Base64.decode</code>	<code>(s -- bytes:ByteList)</code>

```
digest Hex.encode^
digest Base64.encode^
"deadbeef" Hex.decode^
```


SecureRandom

CSPRNG via Go `crypto/rand` — not `math` RNG. Use for nonces, salts, tokens, and key material.

```
// ( n:i -- bytes:ByteList )  
32 SecureRandom.bytes^  
16 SecureRandom.bytes^    // e.g. Argon2id salt
```

`Cipher.generateNonce^` draws a nonce of the correct length for that AEAD via the same source.

Keys

Key is opaque, algorithm-tagged material. Capabilities depend on the bound algorithm.

Constant		generate [^] leaves	Notes
Key.aes256gcm		one Key (32 bytes)	feeds Cipher.aes256gcm
Key.chacha20poly1305		one Key (32 bytes)	feeds Cipher.chacha20poly1305
Key.ed25519		priv then pub	.sign [^] / .verify [^]
Key.x25519		priv then pub	.agree [^] → shared ByteList
Kind	Name	Effect	
class	generate	(algo -- key) or (algo -- priv pub)	
class	fromBytes	(algo bytes -- key)	
class	fromPublic	(algo bytes -- key) public half (Ed25519 / X25519)	
method	toBytes	(-- bytes:ByteList)	
method	sign	(msg -- sig:ByteList) Ed25519 private	
method	verify	(msg sig -- bool) Ed25519 public	
method	agree	(peerPub:Key -- shared:ByteList) X25519 private	

fromBytes : symmetric 32 bytes; Ed25519 64 = private, 32 = public; X25519 32 = private (use fromPublic for public).

Ed25519

```
Key.ed25519 Key.generate^ >ys >xs    // y:pub x:priv
msg x:.sign^ >ts
msg t:peek y:.verify^                // bool
```

X25519

```

Key.x25519 Key.generate^ >ys >xs    // y:pubA x:privA
Key.x25519 Key.generate^
>ts                                  // t:pubB
drop                                  // drop privB if unused
t:peek x:.agree^                     // shared:ByteList – feed Kdf, then
Cipher

```

See `examples/crypto/1a-seal-and-sign.fnl` and `1c-derive-key.fnl`.

Ciphers (AEAD)

Cipher binds algorithm + matching **Key**. Encrypt/decrypt take a **params DynamicObject**.

Constant		Notes
<code>Cipher.aes256gcm</code>		Prefer on hosts with AES-NI; 12-byte nonce
<code>Cipher.chacha20poly1305</code>		Strong default without AES-NI; 12-byte nonce

Kind	Name	Effect
class	<code>new</code>	<code>(algo:s key:Key -- cipher)</code>
method	<code>nonceSize</code>	<code>(-- n:i)</code>
method	<code>generateNonce</code>	<code>(-- nonce:ByteList)</code>
method	<code>encrypt</code>	<code>(params:DO -- ciphertext:ByteList)</code>
method	<code>decrypt</code>	<code>(params:DO -- plaintext true err:s false)</code>

Encrypt fields: `plaintext` (req), `nonce` (req), `aad` (opt).

Decrypt fields: `ciphertext` (req), `nonce` (req), `aad` (opt).

Auth failure: no partial plaintext; modal `false`.

```

: demoAead
  msgBytes >zs
  Key.aes256gcm Key.generate^
  Cipher.aes256gcm swap Cipher.new^ >ts
  t:.generateNonce^ >xs
  { z:peek "plaintext" x:peek "nonce" } t:.encrypt^ >ys
  { y:peek "ciphertext" x:peek "nonce" } t:.decrypt^
  nif
    "decrypt failed: " $ $ nl
    error
  then
    // plaintext:ByteList
    t:drop x:drop y:drop z:drop
;

```

Same API for ChaCha20-Poly1305: `Key.chacha20poly1305` + `Cipher.chacha20poly1305`.

Full example: `examples/crypto/1a-seal-and-sign.fnl`.

Mac (HMAC)

Keyed authenticity **without** encryption. Distinct from Hash and Kdf. One-shot only.

	Constant	Algorithm
	Mac.hmacSha256	HMAC-SHA-256
Method	Effect	
Mac.hash	(algo key bytes -- tag:ByteList)	
Mac.verify	(algo key bytes tag -- bool) constant-time	

Key may be **ByteList**, **String**, or **Key** (uses key material).

```
Mac.hmacSha256 "demo-hmac-secret" msgBytes Mac.hash^ >ts
Mac.hmacSha256 "demo-hmac-secret" msgBytes t:peek Mac.verify^ // true
Mac.hmacSha256 "demo-hmac-secret" "tampered" t:peek Mac.verify^ // false
```

Example: `examples/crypto/1b-hmac-tag.fnl`. Prefer [Env / Moniker](#) for real shared secrets.

Kdf

Derive key bytes from IKM or a passphrase. Leaves `ByteList` — wrap with `Key`.
`<algo> ... Key.fromBytes^` for Cipher.

Constant	Role
<code>Kdf.hkdfSha256</code>	After X25519 agree → AEAD key
<code>Kdf.argon2id</code>	Passphrase → key

```
// ( algo:s params:D0 -- bytes:ByteList )
Kdf.hkdfSha256 params Kdf.derive^
```

HKDF-SHA-256 params

Field	Required	Notes
<code>ikm</code>	yes	ByteList String
<code>length</code>	yes	Int (output bytes)
<code>info</code>	no	default empty
<code>salt</code>	no	optional

```
{ shared "ikm" "final-aead" "info" 32 "length" }
Kdf.hkdfSha256 swap Kdf.derive^
Key.aes256gcm swap Key.fromBytes^
```

Argon2id params

Field	Required	Default if omitted
<code>password</code>	yes	—
<code>salt</code>	yes (≥ 8 bytes)	—
<code>length</code>	yes	—
<code>time</code>	no	3
<code>memoryKiB</code>	no	65536 (64 MiB)
<code>parallelism</code>	no	4

```
16 SecureRandom.bytes^ >xs
{
  "demo-passphrase" "password"
  x:peek "salt"
  32 "length"
}
Kdf.argon2id swap Kdf.derive^
```

Demos may lower `time` / `memoryKiB` / `parallelism` for speed; production should omit them (safe defaults) or set higher costs.

Example: `examples/crypto/1c-derive-key.fnl` (X25519 → HKDF → AES-GCM, plus Argon2id).

Linear Algebra: Vectors, Matrices, Stats, Model Fitting

Final includes a small **dense float64** linear-algebra API for data you already hold as numbers: a vector type, a rank-2 matrix type, stack operators on those values, and ordinary least-squares (OLS) fit / compare / predict.

This is **not** a growable List. `VecNf` and `Mat2f` have a length (and matrix shape) fixed at construction. Use `FloatList` when you need append, insert, or a general numeric sequence; convert at the boundary when you are ready to compute.

Page	Contents
Vectors and Matrices	<code>VecNf</code> / <code>Mat2f</code> : create, fill, index, iterate
vec:: and mat:: Operators	First-wave arithmetic, product, solve, row/col, copy
OLS Fit, Compare, Predict	<code>OLSModel</code> / <code>ModelCompareResult</code> — no factory; materialized by <code>fit</code>

Type `help VecNf`, `help Mat2f`, `help vec::dot`, `help mat::*`, or `help OLSModel` in the TUI for the same stack effects inline.

A worked data story that uses these types is the [Auto-stats](#) sample (used-car asking price, PRE/F model comparison).

Pattern

```
3 VecNf.new^ [ 1. 2. 3. ]
2 2 Mat2f.new^ [ 1. 0.  0. 1. ]
mat::matvec      // ( A:Mat2f v:VecNf -- w:VecNf )
```


Vectors and Matrices

VecNf is a dense vector of **Float** (IEEE-754 float64) of length **N**, chosen when you call `.new^`. **Mat2f** is a dense rank-2 matrix of the same cell type, with shape **rows** × **cols** fixed at construction. Both start **zero-filled**.

They are **not** Lists: there is no `append`. Index stores require a valid slot and a **Float** — Final does not coerce Int into these cells.

Filling from a stack of values uses the `[ ... ]` filler (the closer does not create the instance). See [Collection Builders](#) for nesting rules.

Related: [vec:: and mat:: Operators](#), [OLS](#).

VecNf

Kind	Name	Effect	Description
class	<code>new</code>	<code>(n:i -- v:VecNf)</code>	zero-filled vector; <code>n</code> must be > 0
prop	<code>len</code>	<code>(-- len:i)</code>	number of cells

Indexed access is the List-style ordinal pair:

```
cs:@()  // ( index:i -- f:Float )
cs:!( )  // ( val:Float index:i -- )
```

```
3 VecNf.new^ >ts
[ 1. 2. 3. ]
0 t:@() $ n!           // 1
9. 1 t:!( )           // replace index 1
t:drop
```

`foreach` walks cells in order `0 ... len-1`. The loop value `i` is each **Float**.

Mat2f

Kind	Name	Effect	Description
class	<code>new</code>	<code>(rows:i cols:i -- m:Mat2f)</code>	zero-filled; both dimensions > 0

Kind	Name	Effect	Description
prop	rows	(-- rows:i)	row count
prop	cols	(-- cols:i)	column count
prop	len	(-- len:i)	rows * cols
method	getRowIter	(-- rows:Range)	row indexes 0 ... rows-1
method	getColIter	(-- cols:Range)	column indexes 0 ... cols-1
method	wrap	(row:i col:i -- row:i col:i)	torus wrap of coordinates

Two index styles:

Affix	Stack	Meaning
@() / ! ()	(index:i -- f) / (f index:i --)	flat row-major index 0 ... len-1
@[] / ! []	(r:i c:i -- f) / (f r:i c:i --)	coordinates (row, col)

Teaching line: @() = flat; @[] = coordinates.

```
2 3 Mat2f.new^ >ts
[
  0. 1. 2.
  3. 4. 5.
]
1 2 t:@[] $ nl          // 5 (row 1, col 2)
9. 0 0 t:![[]          // put 9 at (0,0)
t:drop
```

`foreach` on a `Mat2f` walks **flat** cells in row-major order; `i` is each **Float**. Nested `range` loops plus `@[]` walk by coordinates (`i` is the inner index, `j` the outer):

```

: showMat
2 2 Mat2f.new^ [ 1. 2. 3. 4. ] >ts
0 t:.rows@ 1 - range
foreach
  0 t:.cols@ 1 - range
  foreach
    j i t:@[] $ sp
  /foreach
  nl
/foreach
t:drop
;

```

`.getRowIter^` / `.getColIter^` return a `Range` over indexes if you do not want to write `0 .rows@ 1- range` by hand. Pull a whole row or column as a new `VecNf` with `mat::row` / `mat::col` — [operators](#).

Notes

- Every `[ ... ]` fill must supply **exactly** `.len` values, each already a Float.
- Prefer `VecNf` / `Mat2f` when every cell is a Float and the shape is known (design matrix **X**, outcome **y**, coefficients). Prefer numeric Lists when the sequence still needs to grow.
- `help VecNf` / `help Mat2f` lists members and accessors.

vec:: and mat:: Operators

Linear-algebra operators live in two families, like `str::` and `dt::`. **Last argument is TOS.** They return **new** vectors or matrices; inputs are not mutated.

There is no element-wise (Hadamard) multiply in this wave. `vec::dot` is the scalar inner product. `mat::*` is matrix product, not cell-wise `*`.

Type `help vec::+` or `help mat::solve` in the TUI for stack effects.

vec:: (vectors only)

Word	Effect	Description
<code>vec::+</code>	<code>(u:VecNf v:VecNf -- w:VecNf)</code>	element-wise sum; same length
<code>vec::-</code>	<code>(u:VecNf v:VecNf -- w:VecNf)</code>	element-wise $u - v$; same length
<code>vec::scale</code>	<code>(v:VecNf s:f -- w:VecNf)</code>	multiply every cell by <code>s</code>
<code>vec::dot</code>	<code>(u:VecNf v:VecNf -- f)</code>	inner product $\rightarrow$ Float; same length
<code>vec::norm</code>	<code>(v:VecNf -- f)</code>	Euclidean (L2) norm
<code>vec::copy</code>	<code>(v:VecNf -- w:VecNf)</code>	deep copy

```
3 VecNf.new^ [ 1. 2. 3. ]
3 VecNf.new^ [ 4. 5. 6. ]
vec::+ >ts
0 t:@() $ nl          // 5
t:drop
```

```
3 VecNf.new^ [ 1. 2. 3. ]
3 VecNf.new^ [ 4. 5. 6. ]
vec::dot $ nl          // 32
```

mat:: (at least one matrix)

Word	Effect	Description
<code>mat::+</code>	<code>(A:Mat2f B:Mat2f --)</code>	element-wise sum; same shape

Word	Effect	Description
	<code>C:Mat2f)</code>	
<code>mat::-</code>	<code>(A:Mat2f B:Mat2f -- C:Mat2f)</code>	element-wise $A - B$; same shape
<code>mat::scale</code>	<code>(A:Mat2f s:f -- - C:Mat2f)</code>	multiply every cell by <code>s</code>
<code>mat::*</code>	<code>(A:Mat2f B:Mat2f -- C:Mat2f)</code>	matrix product; <code>A.cols</code> must equal <code>B.rows</code>
<code>mat::matvec</code>	<code>(A:Mat2f v:VecNf -- w:VecNf)</code>	$w = A v$; <code>v.len</code> must equal <code>A.cols</code>
<code>mat::transpose</code>	<code>(A:Mat2f -- At:Mat2f)</code>	shape becomes <code>cols × rows</code>
<code>mat::solve</code>	<code>(A:Mat2f b:VecNf -- x:VecNf)</code>	solve $A x = b$ (square or least-squares); <code>b.len = A.rows</code> ; result length <code>A.cols</code>
<code>mat::row</code>	<code>(A:Mat2f r:i -- - v:VecNf)</code>	copy of row <code>r</code> (length = cols)
<code>mat::col</code>	<code>(A:Mat2f c:i -- - v:VecNf)</code>	copy of column <code>c</code> (length = rows)
<code>mat::identity</code>	<code>(n:i -- I:Mat2f)</code>	$n \times n$ identity; $n > 0$
<code>mat::copy</code>	<code>(A:Mat2f -- B:Mat2f)</code>	deep copy

```

2 2 Mat2f.new^ [ 1. 2.  3. 4. ]
2 VecNf.new^ [ 1. 0. ]
mat::matvec >ts
0 t:@() $ sp 1 t:@() $ nl      // 1 3
t:drop

2 mat::identity
2 2 Mat2f.new^ [ 1. 2.  3. 4. ]
mat::*                      // same as the right-hand matrix

```

Mismatched lengths or shapes **panic** (programmer error: the sizes were known when you built the values).

For fitting $y \approx x \beta$ and comparing nested models, use [OLS](#) rather than assembling `mat::solve` by hand — OLS also reports SSE, R^2 , RMSE, and compact-vs-augmented PRE/F.

OLS Fit, Compare, Predict

OLS (ordinary least squares) fits $y \approx X\beta$ for a dense outcome vector and design matrix. There is **no factory** on `OLSModel`. The class method `fit` materializes a model from `y` and `x`.

`x` is $n \times p$ (rows = observations). You supply the intercept yourself if you want one — typically a first column of 1. `y` must have length `n`. Fit requires $n \geq p$.

A compact-vs-augmented comparison (same `y`, nested predictors) follows Judd–McClelland–Ryan: **PRE** (proportional reduction in error) and an **F** statistic. See [Auto-stats](#) for a full ladder on real listings.

Type `help OLSModel` and `help ModelCompareResult` in the TUI.

OLSModel

Kind	Name	Effect	Description
class	<code>fit</code>	<code>(y:VecNf X:Mat2f -- model:OLSModel)</code>	least-squares fit
class	<code>compare</code>	<code>(c:OLSModel a:OLSModel -- r:ModelCompareResult)</code>	compact vs augmented (same <code>n</code> ; <code>a.p > c.p</code>)
method	<code>predict</code>	<code>(Xnew:Mat2f -- yHat:VecNf)</code>	apply β to new rows; <code>Xnew.cols</code> must equal <code>p</code>
prop	<code>beta</code>	<code>VecNf</code>	coefficients, length <code>p</code>
prop	<code>yHat</code>	<code>VecNf</code>	fitted values, length <code>n</code>
prop	<code>sse</code>	<code>Float</code>	residual sum of squares
prop	<code>sst</code>	<code>Float</code>	total sum of squares about the mean of <code>y</code>
prop	<code>r2</code>	<code>Float</code>	$1 - sse/sst$ (0 if <code>sst</code> is 0)

Kind	Name	Effect	Description
prop	rmse	Float	$\sqrt{(sse / (n - p))}$; NaN if $n - p \leq 0$
prop	n	Int	observations
prop	p	Int	columns of x
prop	df	Int	$n - p$

```

4 VecNf.new^ [ 1. 3. 4. 8. ]
4 2 Mat2f.new^ [
  1. 0.
  1. 1.
  1. 2.
  1. 3.
]
OLSModel.fit^ >ts
t:.r2@ $ nl
t:.beta@ >ys
0 y:@() $ sp 1 y:@() $ nl
y:drop
// score a new row (intercept, x)
1 2 Mat2f.new^ [ 1. 2. ]
t:.predict^ >ys
0 y:@() $ nl
y:drop
t:drop

```

Park the model on **ts** (or another aux stack) when you will read several properties or call `.predict^`.

ModelCompareResult

Build two models on the **same** y : a **compact** model **C** and an **augmented** model **A** that adds predictors. Then:

```

mC mA OLSModel.compare^ >ts
t:.pre@ $ sp t:.f@ $ nl
t:drop

```

PRE is $(SSE_C - SSE_A) / SSE_C$ — the fraction of C's leftover error that A removes. **F** asks whether that PRE is large relative to the extra parameters and residual df.

Prop	Type	Description
<code>sseC / sseA</code>	Float	residual SS of C and A
<code>pre</code>	Float	proportional reduction in error
<code>f</code>	Float	F statistic
<code>df1</code>	Int	extra parameters (<code>pA - pC</code>)
<code>df2</code>	Int	residual df of A (<code>n - pA</code>)
<code>n</code>	Int	shared observation count
<code>pC / pA</code>	Int	column counts

Compare **panics** if the models are not nested in this sense (different `n`, or `pA` not greater than `pC`). That is a modeling mistake, not a `throw`.

Notes

- Put **Float** in every design-matrix cell. Int literals in `[ ... ]` are not coerced.
- `fit / predict / compare` panic on nil, shape mismatch, or `n < p` — fix the matrices; do not wrap these in `try` for “maybe the data is the wrong width.”
- For extracting a SQL result into a List or Table first, see `iter>list` and `dbIter>table`.

Raster Drawing

Final provides a rich complement of features that support raster-oriented drawing. It includes the following main objects and operations:

- 1, 4, and 8 bit color index matrices
- 1, 4, and 8 bit Palettes
- a full Color matrix
- Pixmap, an image drawing surface
- 1, 4, 8 bit, and full-color (RGBA) blitting operations
- Drawing operations that include lines, rectangles, circles, and many others
- import/export from/to PNG JPG and GIF

All the features above are available in every version of final. The non-gui versions will primarily use these features for image file generation, processing, rendering, and web serving.

The `final-gui` version of Final includes support for the [The Fyne UI Library](#), with support for Windows, Canvas, a variety of widgets such as text entry, radio, select, etc. It also includes a variety of layout containers to conveniently lay out an interface. A Pixmap can be set into a CanvasImage widget for direct visualization.

Raster Blitting

Raster operations are of two basic types:

- pixel and pixel mat operations called “blitting” (Bit BLock Transfer)
- drawing, such as lines, shapes, and text

Final provides a helpful assortment of objects and operations to facilitate both types of operations.

The best way to understand these objects is to read the source code where most of them are put to use, in `examples/blit-draw` and `examples/gui/simgame` (Life, Shape Zombies, and Invaders).

Coordinate System

The coordinate system for both blitting and drawing follows the longstanding game convention of:

- $x=0$ is the far right, and increases positive to the right
- $y=0$ is the top left, and increases positive going down

This is a bit different from mathematic coordinates, but it does correspond to things like spreadsheets.

Blitting and matrix access is always done from a reference point of top-left, down to bottom right.

Most painting operations involving shapes use a center-0 reference, such as the center of a circle or rectangle.

`Pixmap` supports a world transformation `viewportAt` property which enables you to locate the viewport anywhere in an unconstrained drawing space; blits and draws clip automatically.

Utility Types

There are several utility types used widely across drawing and painting. They are briefly described here. All these types are immutable value types, meaning they can’t be modified after creation, and are passed by value, not by pointer. You create a new one when you need new values. Use `help` to view the members and factories provided by each type.

Point

A 2-dimensional point, using fractional x and y.

Member	Stack effect	Description
<code>.x</code> <code>.y</code>	<code>(-- n:f)</code>	exact coordinates
<code>.xi</code> <code>.yi</code>	<code>(-- n:i)</code>	rounded integral coordinates
<code>Point.new</code>	<code>(x:n y:n -- p:Point)</code>	create a new Point; accepts any numeric type
<code>Point.zero</code>	<code>(-- p:Point)</code>	the 0,0 Point (prefer this to the <code>new</code> factory for known 0,0)

Note that most APIs that take a Point in Final will also accept the `( x:n y:n - )` shape.

Also note that x and y are reversed, coordinate-wise, from the row/col parameters used in things like Mats, so be wary!

Size

A 2-dimension size, using Float w (width) and h (height).

Member	Stack effect	Description
<code>.w</code> <code>.h</code>	<code>(-- n:f)</code>	exact width and height
<code>.wi</code> <code>.hi</code>	<code>(-- n:i)</code>	rounded integral width and height
<code>Size.new</code>	<code>(w:n h:n -- s:Size)</code>	create a new Size; accepts any numeric type

Note that most APIs that take a Size in Final will also accept the `( w:n h:n -- )` shape. Discrete sinks such as `Pixmap.new` and `Painter.new` require exact integral dimensions.

Rect

Describes a rectangular region; comprises a Point topLeft and a Size size.

Member	Stack effect	Description
<code>.topLeft</code>	<code>(-- p:Point)</code>	top-left corner
<code>.size</code>	<code>(-- s:Size)</code>	extent (W×H)

Member	Stack effect	Description
<code>Rect.new</code>	<code>(topLeft:Point x:n y:n size:Size w:n h:n -- r:Rect)</code>	create a <code>Rect</code> ; each slot accepts a value type or a numeric pair

Examples of the modal factory shapes:

```
pt sz Rect.new^
x y sz Rect.new^
pt w h Rect.new^
x y w h Rect.new^
```

Blitting

Blitting, stated simply, is moving pixels from A to B, typically from some source, to in Final a destination called a `Pixmap`. A `Pixmap` is like a canvas of pixels, upon which you successively build up a final image from a series of simpler primitives. `Pixmap` is used for both Blitting and Drawing, and the two types of operations can be intermixed, with some caveats. A `Pixmap` is created with a specific size, which is immutable for its lifetime; so the canvas size must be set before any drawing operation commence.

The following sections explain the various “atoms” of the blitting capabilities

Color

A color object is a 32 bit composite of 8 unsigned bits each for Red, Green, Blue, and Alpha. Alpha is how transparent or opaque is the color value, with 0 being completely transparent (and thus not really a color—the other 3 values are irrelevant) and 255 being completely opaque. When you do not specify an A value for a `Color` it is automatically set to be 255/opaque.

Alpha is not meaningful in isolation; it only makes sense when drawing upon a `Pixmap` that may already have had drawing operations made on it. In this case, if using the `blend` operation, Alpha will determine how much of the new color to mix with the existing values. (Of course opaque stomps over top.)

The `Color` object has constants for some common colors from the Web Palette, and also `translucent` and `mask`, which is used in masked blitting operations.

Member	Stack effect	Description
<code>.r .g .b .a</code>	<code>(-- n:i)</code>	Channel values 0–255 (Int); read-only
<code>Color.black ...</code> <code>Color.aqua</code>	<code>(-- c:Color)</code>	Named Web Palette colors (also <code>cyan</code> / <code>magenta</code> aliases for <code>aqua</code> / <code>fuchsia</code>)
<code>Color.transparent</code>	<code>(-- c:Color)</code>	Fully transparent (0,0,0,0)
<code>Color.mask</code>	<code>(-- c:Color)</code>	Same as transparent; skip pixel when blit mask is on
<code>Color.rgb</code>	<code>(r:i g:i b:i -- c:Color)</code>	From 0–255 channels; alpha 255
<code>Color rgba</code>	<code>(r:i g:i b:i a:i -- c:Color)</code>	From 0–255 channels including alpha
<code>Color.rgbf</code>	<code>(r:f g:f b:f -- c:Color)</code>	From 0.0–1.0 channels; alpha 1.0
<code>Color rgbaf</code>	<code>(r:f g:f b:f a:f -- c:Color)</code>	From 0.0–1.0 channels including alpha
<code>Color.hex</code>	<code>(packed:i hex:s -- c:Color)</code>	From <code>0xRRGGBB</code> Int or hex String (optional <code>0x</code> / <code>0X</code>); alpha 255
<code>Color.hexa</code>	<code>(packed:i hex:s -- c:Color)</code>	From <code>0xRRGGBBAA</code> Int or hex String (optional <code>0x</code> / <code>0X</code>)

Palette

A `Palette` is a lookup table, that maps a narrow index value to an actual color. Final supports 2, 4, and 8-bit Palettes. This lets you maintain a source of pixels in a more compact format than full 32-bits, economizing on memory. Although memory is not as scarce as in the 70s and 80s when this technique was widely used, it is still useful in many cases. For example, the `life-sim.fnl` simulation uses 4-bits, 16 values, to encode the age of the cells, and then a palette to hold the actual colors.

The Final types are:

- `Palette1` – 1 bit, 2 color values
- `Palette4` – 4 bits, 16 color values
- `Palette8` – 8 bits, 256 color values

Mat2x

A `Mat` is a tabular structure of a fixed number of rows and columns, established when created. (For details on `Mats`, see [Mats](#)). Four of these types are used in blitting, as shown in this table:

Mat	Palette	Used for
<code>Mat2i1</code>	<code>Palette1</code>	2-value color indexes and lookup
<code>Mat2i4</code>	<code>Palette4</code>	16-value color indexes and lookup
<code>Mat2i8</code>	<code>Palette8</code>	256-value color indexes and lookup
<code>Mat2c</code>	n/a	Full color operations, import/export

`Mats` in Final have matrix compound path accessors:

- `@[]` (row:i col:i – value) for fetch
- `![]` (value row:i col:i –) for store

They also have list-like flat-index accessors; storage is in row-major order.

- `@()` (flat:i – value) for fetch
- `!()` (value flat:i –) for store

Flat indexes are useful for setting the entire matrix to a specific value, or sometimes when assigning initial randomized values that are unrelated to row/col.

BlitSrc

A `BlitSrc` is the source data of a blitting operation. It always holds one of the four `Mat` types, and for the Palettized ones (1/4/8) the accompanying `PaletteX`. (`Mat2c` does not use a `Palette`.) You provide a `BlitSrc` to the `blit` method of `Pixmap`.

Pixmap

A `Pixmap` is a rectangular blitting and drawing surface whose size is set at creation. Its underlying representation differs from `Mat*`, but accessing individual pixels uses the same operators. (The underlying representation is based on what the core code for blitting and drawing are most efficient using.)

A typical session with a `Pixmap` involves creating, optionally setting a background color with `fill`, then performing a series of `blit` or draw operations. Successive operations either write over and blend their content with the existing state of the `Pixmap`.

A `Pixmap` can be exported to a PNG/JPEG/GIF file, and can be serialized using a `Stream` for Http operations. It can also be directly visualized using the `Fyne CanvasImage` object (for final-gui binary.)

`Pixmap` has several blitting options, which can be set on the `Pixmap`, and then optionally overridden for an individual `blit` call:

Option	Values	Default	Description
mask	on or off	off	when on, uses the 0 palette index, or for Mat2c the <code>mask</code> color to indicate pixels that should not be written to the <code>Pixmap</code>
op	stomp or blend	stomp	stomp overwrites the destination pixel; blend mixes according the alpha values of the src and dst colors
cellPx	Int	1	how many dst pixels to write, per source pixel see <code>life.fnl</code> for an example, where it uses 4 to make each cell a good size

Drawing

Final provides a set of flexible and powerful drawing operations for things like lines, shapes, and text. It provides options for stroke (lines/shape outlines) and fill (the area inside shapes). Text can be aligned vertically and horizontally to suit the context. A `push` and `pop` facility enables preserving previously set parameters when drawing a set of new values; this is useful when calling higher level drawing constructs composed from many primitives.

Painter

Drawing is exposed on the `Painter` object, which can take an existing `Pixmap` or can create one automatically. Therefore, blitting and drawing operations can be mixed in any sequence, keeping in mind that new operations work over top of previous ones. The api is briefly summarized below. All the available operations are used in the `examples/blit-draw/draw.fnl`. Painting is also used in the Shape Zombies and Invaders example applications.

As with blitting, a Painter has a default set of parameters, which can be overridden. Unlike Pixmap, the mechanism of overriding is not on a per api basis, but by way of using `push`, setting the new parameters, using them, then calling `pop`, which restores the previous values.

Use `help Painter` for details on the parameter values needed for each operation. “Rot” indicates an operation that rotates the shape clockwise around its center by the specified degrees.

API	Description
<code>pixel</code>	write a pixel with current <code>stroke</code> value
<code>pixelC</code>	write a pixel with a provided Color
<code>line</code>	Draw a line
<code>circle</code>	Draw a circle
<code>ellipse</code> <code>ellipseRot</code>	Draw an ellipse
<code>rect</code> <code>rectRot</code>	Draw a rectangle
<code>roundedRect</code> <code>roundedRectRot</code>	Draw a rounded rectangle

API	Description
<code>regularPolygon</code> <code>regularPolygonRot</code>	Draw a regular polygon
<code>text</code> <code>textWrap</code>	Write text, auto-wraps \n; <code>textWrap</code> wraps to a rectangular area

Text writing can be done using the following alignments:

- Vertical: Top, Middle, Baseline (line above descenders such as in “g”), Bottom
- Horizontal: Left, Center, Right

Pens stay on the Painter (`stroke` , `fill` , `font`). `push` / `pop` are a pen bracket around whole sessions, not part of the builder.

Contours that are not a stock shape use `pathBuilder` : a singleton session on the Painter.

PathBuilder

You can use Painter to create paths and solids of arbitrary strokes, using straight lines, bezier curves, arcs, or elliptical curves. These all use a `PathBuilder` .

`Painter.pathBuilder` is created with the Painter. Note that it is **not** in `PainterState` and is not cloned on `push` . It is one of three states: **ready**, **path**, **solid**.

Word	From	Does
<code>startPath</code>	ready	Open polyline (chart line). <code>end</code> strokes; does not <code>closePath</code> .
<code>startSolid</code>	ready	Closed shape. <code>end</code> closes, then fill then stroke.
<code>moveTo</code> / <code>lineTo</code> / <code>bezierTo</code> / <code>arc</code> / <code>ellipticalArc</code>	path or solid	Append. Point or <code>x y</code> (Int, UInt, or Float).
<code>end</code>	path or solid	Paint, clear the gg path, back to ready.

Do **not** `lineTo` the start point on a solid; `end` closes.

One-shots that would wipe the contour (`line` , `circle` , `ellipse` , `ellipseRot` , `rect` , `rectRot` , `roundedRect` , `roundedRectRot` , `regularPolygon` , `regularPolygonRot`) and

`push` / `pop` **panic** unless the builder is **ready**. `pixel` / `pixelC` / `text` / `clear` do not use the path and stay allowed.

Invalid combinations panic with the state in the message (`already in path`, `not started`, `call end first`).

Good

Park the builder, start, emit, `end`, drop. Mix one-shots only while **ready**. Reuse the same object after `end`.

```
// Open stroke (chart / polyline)
p.pathBuilder >ts
  t:.startPath^
  10 10 t:.moveTo^
  80 10 t:.lineTo^
  80 80 t:.lineTo^
  t:.end^
t:drop

// Closed fill + stroke – do not repeat the start point
FillStyle.solid p.fill.style!
Color.aqua p.fill.color!
p.pathBuilder >ts
  t:.startSolid^
  16 16 t:.moveTo^
  80 16 t:.lineTo^
  48 72 t:.lineTo^
  t:.end^
t:drop

// Fractional control points (flex numeric)
t:.startPath^
10 40 t:.moveTo^
40.5 8 90 40 t:.bezierTo^
t:.end^
```

Don'ts

`p.line` (or `circle` / `ellipse` / `rect` / `roundedRect` / `regularPolygon` / `*Rot` / `push`) after `startPath` and before `end`. **Don't:** `end` with no `start` *. **Don't:** `startSolid` while already in **path**.

Temporary pens for a series of sessions: `push`, set `stroke` / `fill`, several `start*` ... `end` (and one-shots), then `pop`.

Reading/Writing to files or streams

Both `Mat2c` and `Pixmap` have methods for saving to PNG, which is lossless, or JPEG/GIF which are lossy. It is recommended to using PNG.

They also both have a factory method for creating an instance from a file. The `newFromFile` api automatically determines the file format.

The Fyne UI Library

[Fyne](#) is a Go-based cross-platform Graphical User Interface framework. Final uses this framework for the final-gui interface, and for all of its GUI components.

This section introduces you to the Fyne Library, and how Final presents it to you. The next chapter covers the basic concepts of the library, the following chapter briefly covers the Fyne widgets, layouts, and helper objects available in Final.

The [SimGame Framework](#) also uses the Fyne library, as well as Final's [Raster Drawing](#) framework.

Future releases of Final are likely to expand Fyne support.

Fyne and final-gui

Fyne is a library of Graphical Interface components, and helper classes, similar in design to many other such libraries. You compose an interface with a combination of layout components, and individual widgets.

There are two central components to any interface you might build:

- `App` which is the top-level Fyne object; you access this in Final via `App.app`.
- `Window` which is the root object that contains all your other elements. Individual widgets can only be shown once added to a Window, via its `setContent` method, which must either be a Layout or Widget.

There are numerous examples of how to use Fyne in `examples/gui/fyne`. Here is a simple “Hello Fyne!” demo that displays the venerable message in a Label widget, then updates the text four seconds later. It shows how easy it is to set up and display a Fyne Window.

```
: hello-fyne
App.app "Hello Fyne Application" Window.new^ >ts

"Hello from Fyne!" Label.new^ dup >xs
t:.setContent^

t:.show^

4000 delayMs
"See ya later Fynagator!" x:.setText^

x:drop t:drop
;
```

Layouts

A `Layout` is a component that helps organize the widgets it contains, so you don’t have to try manually setting the position of the widgets, which can be tedious, and fragile when you make changes.

Fyne has simple layouts, like a horizontal and vertical Box, which layout the widgets in those orders. It has a Form layout, for aligning labels with widgets in a grid. It has a sophisticated Border layout, which can place top/bottom/left/right/ and a central widget on the screen.

Widgets

Final supports a variety of Fyne's widget types, such as Label, Entry, Button, and so on. These are covered in detail in the next chapter.

Menus

Fyne supports menu bars, menus, and menu items. The Fyne app `examples/gui/fyne/widgets-fyne.fnl` shows how to add a menu bar to your window.

Events

Fyne provides change notification, such as Tapping a button, typing in an Entry field, and selecting a menu item via Events, the basics of which were covered in [Events and EventLoop](#). You create an event handler in a Final method on a TypedObject instance, then use a MethodRef to obtain a reference to that method, which, along with an index, you pass to a Fyne api that requires or accepts an Event Handler. When the event is triggered, your handler Method runs, with your TypedObject instance on the O Stack, where you can read/write its properties, or call other methods or helper words.

Preferences

Fyne manages a local storage location into which you can save and fetch preferences associated with your app by using the Prefs class object.

Fyne Controls

This chapter is a succinct summary of the Layouts and Widgets available to use in Final. Most of these components are included in the example program

`examples/gui/fyne/widgets-fyne.fnl`, which you are encouraged to peruse to see how to use each one. The SimGame framework examples also make extensive use of Fyne Events.

Most layouts and widgets also support `Canvas.visible` and `Canvas.refresh`.

Interactive fields such as Button, Entry, Check, Select, RadioGroup, Slider, and Form also support `Disableable.disabled`.

Layouts

A layout arranges the widgets it contains so you do not have to place them by hand. Box, Border, Grid, and FormLayout are factories that return a `Container`. Accordion, AppTabs, Scroll, and Split are layout types you create and keep as instances.

Layout	Role
Accordion	Stack of titled sections that open to show detail
AppTabs	Row of titled tabs, each with its own content
Border	Five slots: top, bottom, left, right, and center
Box	Horizontal row or vertical stack
Container	Layout object returned by Box, Border, Grid, and FormLayout
FormLayout	Two-column label/widget pairs (no Submit/Cancel)
Grid	Equal cells in a fixed number of columns or rows
Scroll	Scrolling viewport around one child
Split	Two panes with a movable divider

Accordion

A stack of titled sections. Create an empty Accordion, then `append` each title and its detail content.

```
Accordion.new^ >ts
"Item 1" "Detail one" Label.new^ t:.append^
"Item 2" "Detail two" Label.new^ t:.append^
ts>
```


AppTabs

A row of tabs. Create empty, then `append` each title and content. Read or set `selectedIndex` (0-based).

```
AppTabs.new^ >ts
"Basic"  o:..makeBasicTab^  t:..append^
"Border" o:..makeBorderTab^ t:..append^
ts>
```

Border

Places up to five children: top, bottom, left, right, and center. Pass `null` for any unused slot. The center receives leftover space, which is useful for a Split or canvas that should grow with the window.

```
"TOP"  Label.new^
"BOTTOM" Label.new^
"LEFT"  Label.new^
"RIGHT" Label.new^
"CENTER" Label.new^
Border.new^
```

Box

`Box.newV` stacks children vertically; `Box.newH` lays them out in a row. Both take a `List` of canvas controls and return a `Container`.

```
(
  "Hello" Label.new^
  "World" Label.new^
) Box.newV^
```

Container

The layout object returned by Box, Border, Grid, and FormLayout. Place it as Window content or nest it inside another layout. Use `Canvas.visible` and `Canvas.refresh` when you need to hide or redraw it.

FormLayout

A two-column form layout: label, then widget, repeated. The list must have an even length. Unlike the Form *widget*, FormLayout has no Submit or Cancel buttons.

```
(
  "Name" Entry.new^
  "Age"   Entry.new^
) FormLayout.new^
```

Grid

Equal-sized cells. `Grid.newColumns` takes a column count and a list of children; `Grid.newRows` takes a row count.

```
2 (
  "A" Label.new^
  "B" Label.new^
  "C" Label.new^
  "D" Label.new^
) Grid.newColumns^
```

Scroll

Wraps one child in a scrolling viewport. Use `setMinSize` when the viewport should stay a fixed size inside a larger layout.

```
(
  "Line 1" Label.new^
  "Line 2" Label.new^
  "Line 3" Label.new^
) Box.newV^ Scroll.new^ >ts
200 120 t:.setMinSize^
ts>
```

Split

Two panes with a movable divider. `Split.newH` places leading and trailing side by side; `Split.newV` stacks top and bottom. Use `setOffset` with a value from 0 to 1 to set the divider position.

```
"Left" Label.new^
"Right" Label.new^
Split.newH^ >ts
0.4 t:.setOffset^
ts>
```

Widgets

Widgets are the interactive or display controls you place inside layouts (or directly as Window content). Event handlers are TypedObject methods; pass a MethodRef (`o: .handlerName~`) and an integer index to the factory or setter that accepts a handler. When the event fires, your instance is on the O stack.

Widget	Role
Activity	Animated busy indicator
Button	Clickable labeled control
CanvasImage	Image on the canvas, with optional mouse handlers
Check	Labeled checkbox
Entry	Single-line or multi-line text field
Form	Labeled fields with optional Submit and Cancel
Label	Static text
ListView	Scrolling list of strings (via ListRef)
RadioGroup	Exclusive radio choices
RichText	Plain or Markdown formatted text
Select	Drop-down list of string options
Separator	Thin dividing line
Slider	Numeric value between min and max
TableView	Grid of cells from a Table (via TableRef)
TreeView	Collapsible tree (via TreeRef / FyneTree)

Activity

An animated busy indicator. Call `start` to run it and `stop` to halt it.

```
Activity.new^ >ts
t:.start^
// ...
t:.stop^
ts>
```

Button

A clickable labeled button. The factory takes a `MethodRef`, an index, and the label text. Use `setText` to change the label later. `Button.newWithIcon` adds a theme icon from the `Icon` class (for example `Icon.Confirm`).

```
// handler: ( ev:Event -- ) o( inst )
: MyApp#onClick
  drop
  "Pressed!" o:.btn@.setText^
;

o:.onClick~ 0 "Click me" Button.new^
```

CanvasImage

Shows a `Pixmap` or `Image` in the window. Call `refresh` after you mutate the underlying `Pixmap`. Optional mouse handlers cover down/up, hover, and scroll; see `examples/gui/fyne/mouse-fyne.fnl`.

```
"photo.png" Image.newFromFile^ CanvasImage.new^ >ts
// after drawing into a Pixmap shared with this image:
t:.refresh^
ts>
```

Check

A labeled checkbox. Read or write `checked`. Attach `setOnChanged` to learn when the user toggles it; `Event.params[0]` is the new `Bool`.

```
"Enable" Check.new^ >ts
o:.onCheckChanged~ 0 t:.setOnChanged^
ts>
```

Entry

A text field. `Entry.new` is single-line; `Entry.newMultiLine` wraps. Read or write `text`. Use `setOnChanged` while typing and `setOnSubmitted` when the user presses Enter on a single-line Entry; both deliver the String in `Event.params[0]`.

```
Entry.new^ >ts
o:.onEntryChanged~ 0 t:.setOnChanged^
o:.onEntrySubmitted~ 0 t:.setOnSubmitted^
ts>
```

Form

A labeled list of fields with optional Submit and Cancel. `append` each label and widget, then `setOnSubmit` / `setOnCancel`. You can also `showModal` on a parent Window and wait for Ok or Cancel (see Dialog constants `Dialog.btnOk` / `Dialog.btnCancel`).

```
Form.new^ >ts
"Name" Entry.new^ t:.append^
"Subscribe" "Yes please" Check.new^ t:.append^
o:.onFormSubmit~ 0 t:.setOnSubmit^
o:.onFormCancel~ 0 t:.setOnCancel^
ts>
```

Label

Static text. Create with `Label.new`, then update with `setText`.

```
"Hello from Fyne!" Label.new^ >ts
"Updated text" t:.setText^
ts>
```

ListView

A scrolling list of strings. Wrap a `StringList` in a `ListRef`, then create the `ListView`. After you change `ListRef.list`, call `ListView.refresh`. `setOnSelected` delivers the row index in `Event.params[0]`.

```
( "Alpha" "Beta" "Gamma" s) ListRef.new^
ListView.new^ >ts
o:.onListSelected~ 0 t:.setOnSelected^
ts>
```

RadioGroup

Exclusive radio choices from a `StringList`. The factory takes an optional `MethodRef` (pass `null` to skip the event), an index, and the options. Read `selected` for the chosen String (or `""` if none).

```
o:.onRadioChanged~ 0
( "One" "Two" "Three" s)
RadioGroup.new^
```

RichText

Formatted text. `RichText.newWithText` takes plain text; `RichText.newFromMarkdown` parses Markdown. Update later with `setText` or `parseMarkdown`.

```
`
# Title

Some bold text.
` RichText.newFromMarkdown^
```

Select

A drop-down of string options. Same factory shape as `RadioGroup`: `MethodRef` (or `null`), index, and a `StringList`. Read `selectedIndex` (0-based). The change handler receives the selected String in `Event.params[0]`.

```
o:.onSelectChanged~ 0
( "Red" "Green" "Blue" s)
Select.new^
```

Separator

A thin dividing line between other controls.

```
(
  "Above" Label.new^
  Separator.new^
  "Below" Label.new^
) Box.newV^
```

Slider

A numeric slider. Create with min, max, and step; read or write `value`. `setOnChanged` fires while dragging; `setOnChangeEnded` fires when the drag ends. Both deliver a Float in `Event.params[0]`.

```
0 100 1 Slider.new^ >ts
o:.onSliderChanged~ 0 t:.setOnChanged^
50 t:.value!
ts>
```

TableView

A grid of cells from a Final `Table`. Wrap the table in a `TableRef`, then `TableView.new` with a `Bool` for whether to show column-name headers. Use `setColumnWidth` or `setColumnWidths` for sizing. After changing the table data, call `refresh`. Selection handlers receive row then column in `Event.params`.

```
myTable TableRef.new^
true TableView.new^ >ts
( 120 60 i) t:.setColumnWidths^
o:.onTableSelected~ 0 t:.setOnSelected^
ts>
```

TreeView

A collapsible tree. Build a `FyneTree` (root id 0), add branch and leaf nodes, wrap it in a `TreeRef`, then create the `TreeView`. Selection delivers the node id in `Event.params[0]`. Refresh after changing the tree data.

```
FyneTree.new^ >ts
t:.root@ >xs
x:peek "Animals" t:.newBranchNode^ >ys
y:peek "Dog" t:.newLeafNode^ drop
y:drop x:drop
ts> TreeRef.new^
TreeView.new^ >ts
o:.onTreeSelected~ 0 t:.setOnSelected^
ts>
```

Simulations and Games

Final provides a framework called `SimGame` to help create Simulations and Games. There are several example programs that exercise most of the functionality of the framework, and can serve as a reference or template for new sims and games:

- `life.fnl` – Conway’s Game of Life – a classic and beguiling cellular automata simulation
- `shape-zombies.fnl` – Shape Zombies - a simple but surprisingly fun game of hitting Bless circles and avoiding Zombie squares
- `invaders.fnl` – Invaders – a simple take on the classic Space Invaders, that automatically gets harder the better you perform

NOTE: the `SimGame` framework uses the Fyne GUI library and is only available when running `final-gui`.

The framework provides the following constructs and facilities:

SimGame

`SimGame` is the top-level organizational/containing type for any simulation or game it provides:

- creation of the GUI, with hooks for adding a menu and bottom bar/area of any content
- a hybrid composition of a Final catalog type with your own `TypedObject` for holding your game objects and state, and writing handler methods for things like rendering, key/mouse events, etc
- support for keyboard and mouse support
- a Game clock which advances in millisecond increments, and can be used for automatically triggering render and update events
- an optional `MotionHit` engine, described below
- render dispatching of `Drawer` objects you may provide, respecting z-order

Creating a `SimGame` instance involves two steps:

1. create a `TypedObject` that will hold your game state, event handlers, and methods
2. create a `SimGame` object, passing it an instance of your `TypedObject`

The `SimGame` instance will be available to you in all your event handlers and methods as the `.sg` property on your `TypedObject` instance. Likewise, the `SimGame` has your instance, and uses it to send events to.

Coordinates

By default, the game world and viewport are the same, with 0,0 at the upper left, x extending positive right, y down. You can create a `Sim/Game` that is larger than the `Pixmap` used to visualize it, by setting the `Pixmap.viewOrigin` to a non 0,0 point in your world coordinates. The blitting and drawing routines will then accept your world coordinates.

SimGameClock

This clock operates continuously on a background thread, and updates a `SimGame.tick` counter with milliseconds since `Start`. This value becomes a reference for all other time-based operations in the game, and serves as your reference to real time in your updating and rendering handlers. It is used:

- to trigger `update` and `render` events, if those are enabled;
- to cycle animations and effects that have time-based phases;
- to time and dispatch any time-based events you request via `timerEventAfter/At/Every` apis.

Entities

An `Entity` is a visible or invisible object or effect in your game. In *Shape Zombies*, the Entities are the circles, squares, and the user hexagon. In *Invaders*, the Entities are the the star field, aliens, and the cannon. You create an instance of an `Entity` for every instance of a game object. An Entity has these key properties:

Property	Type	Description
<code>data</code>	<code>flex</code>	Defaults to a <code>DynamicObject</code> , but can take any Final type – put your custom Entity data in here
<code>sg</code>	<code>SimGame</code>	the <code>SimGame</code> instance
<code>kind</code>	<code>Int</code>	an ordinal you manage that represents a major kind of Entity Zombies has 3: Bless, Zombie, and User
<code>subkind</code>	<code>Int</code>	you can have subkinds of your Kinds; Zombies used 0/1/2 for the three Bless or Zombie color

Property	Type	Description
		sets
state	Int	an ordinal you can use for a state of the entity, if it has states
at	Point	where the entity is in the game world
hitArea	HitArea	an optional HitArea if the entity will be involved in collisions or boundary actions
drawerSet	DrawerSet	the current visual representation
vx, vy	Float	if it is moving the x and y velocity components, used by Motion engine
zorder	Int	from -ve background to positive, the order things should get drawn (“in front of”/“behind”)
anchored	Bool	a fixed place Entity, or one that does not move when hit
visible	Bool	if it is visible; if not, it is not drawn
enabled	Bool	a “soft switch”; not enabled won’t participate in motion/collisions
active	Bool	is live in the Entities collection; false before adding to game, and when removed

HitAreas

A HitArea defines a spatial extent that is used by the Hit and Boundary collision processes. Every Entity that participates in that process needs a HitArea. Two types are provided:

- CircleHitArea – whose center defaults to the `at` of its Entity, with a defined radius
- RecHitArea – a rectangle whose center defaults to the `at` of its Entity, with defined width/height

A HitArea is not a visible element—it is just an approximate (or exact) outline of the entity. It does not have to exactly correspond to the visible extent of the Entity.

MotionHit Engine

The `MotionHit Engine (MHE)` is a centerpiece of SimGame. It is a background process that moves entities with non-zero vx/vy smoothly in the game space, and

registers “hits” between registered Entities, or between Entities and an enabled boundary. Because it works in the background, on its own thread, it does not take CPU time away from game logic or rendering. It operates precisely, using a high-precision version of game time, and floating point positions and changes. So while you use the simple integral ms game clock at the level of game actions and events, the MHE is operating much more precisely, without “chunking” or rounding.

Action and Event options

All the action and event options of MHE share a common vocabulary, summarized below:

Option	Description
none/off	the option is not used in the context
stop	the entity will stop
bounce	the entity will rebound
events	the defined contact will raise an event

Motion

The engine can move Entities in space, updating their `at` coordinate, according to the velocity set via their `vx/vy` (in pixels/second). Motion does not require HitAreas. See the following sections for boundary and hit processing of moving entities.

You enable Motion by setting these properties (we show typical code contexts)

```
// o( inst:TypedObject )
true o:sg.useMotionEngine!
// entity on t:
<xvalue> t:.vx!
<yvalue> t:.vy!
// OR
<xvalue> <yvalue> t:.setVelocity^
```

That’s it! As long as your entity is `enabled` (default) and not `stopped` the MHE will now move it around, by changing its `at` value. All you need to do is render it using its `at` location in your `handleRender` routine.

Boundary Crossings or Hits

There are two kinds of boundary options:

- `Cross`, based only on the `at` centroid, which doesn't need a `HitArea`
- `Hit`, which is based on an entity's `HitArea` (required for this mode)

The `SimGame` object has separate action and event properties for each of: left, top, right, and bottom boundary. For example, `Invaders` doesn't need its left and right boundaries. There is a convenience operator `setAllBoundardModes` that you can call when the options are the same for all four boundaries. You provide both the `Cross`: none/stop/bounce + event on/off, and `Hit`: none/stop/bounce + event on/off for a boundary. You are free to “mix and match” both types, which will depend on your entity definitions, and whether you use `HitAreas`. You are not required to provide a `HitArea` for all your entities, only the ones you want to participate in the hit-based operations.

For example, just using `at` and `Cross` detection, an Entity can drift partially offscreen until its `at` reaches the boundary; but with `Hit` boundary detection, the entity stops or bounces at its border. An event can be triggered by either type of boundary check, which creates a `bounds` event that you handle. `Invaders` uses this to detect when a Phaser shot leaves the screen at the top, or an alien hits the ground. In both cases it uses the `stop` option, and the event then handles the occurrence, such as removing the phaser shot from the playfield, and scoring a loss for the alien hitting the ground. `Shape Zombies` uses the `bounce` option, so the `Bless` and `Zombie` entities bounce off the boundaries, the same way they bounce off each other.

If you don't want your entity to move when hit, you set the `anchored` property. This just means that it is either a permanent fixture like a wall or barrier, or that the entity doesn't other move when hit.

```
true t:.anchored!
```

Hit Detection

The Hit engine can detect collisions between any two entities with a `HitArea`, or between entities and a boundary, when that boundary's Hit option is `stop` or `cross`. As with `Motion`, an event can be enabled, providing an easy way to process these occurrences, without having to manually iterate your entities, and solve the complex $N \times N - 1$ problem!

HitRegistration

During the init event of your simgame, you register what happens in collisions between pairs of entities, and between an entity and a boundary. The registration uses an integral `HitKind` value, which you define to group similar entities together, and differebt fior collision semantics with other entities gets its own kind.

Unregistered entities simple pass through each other. Registration is transitive, so you only need to register each pair once. The `HitKind` gets set on the `HitArea`. The boundaries have a system-defined `HitKindBoundary` (-1).

Here is the init code for `ShapeZombies` showing all the calls. `y` contains the `SimGame` object

```
// We use Hits to bounce off borders and each other!
true y:.options.useMotionEngine!
true y:.options.useHits!
SimGameOptions.hitActionBounce
y:.options.setAllBoundaryModes^
// Set up the Hit Matrix -- how entities interact
// BZs just bounce off each other; no event
CircZomb.HitKindBZ dup SimGameOptions.hitActionBounce
y:.options.addHitMatrixEntry^
// BZs also bounce off User, but we get an event, for bless/drain
CircZomb.HitKindBZ CircZomb.HitKindUser
SimGameOptions.hitActionBounce SimGameOptions.hitEvents |
y:.options.addHitMatrixEntry^
```

Cyclers, Drawers, and DrawSets

Cycler

A Cycler is an auxiliary object with properties related to cyclical or counted rendering processes.

Property	Type	Description
start	Int	The <code>Simgame.tick</code> when the object was activated – used for cycle/phase calcs
<i>Setup props:</i>		
phaseCount	Int	Phases per cycle; 0 = not phased
phaseRate	Int	ticks per phase; 0 = not phased
phaseOffset	Int	ticks before phase 0 starts; usually 0
cyclesToRun	Int	0 = continuous; else N cycles then done
<i>Dynamic r/o:</i>		

Property	Type	Description
currentPhase	Int	0..phaseCount-1
currentCycle	Int	0..cyclesToRun-1 when finite
cyclesDone	Bool	true after finite run completes; poll in update

Drawer

You can choose to do all your rendering manually. Life and Zombies use this approach. You can also use SimGame’s Drawing Engine, by attaching a DrawerSet to your entities. You can also use a hybrid approach, by letting the DrawingEngine first draw entities with DrawerSets, then your `handleRender` event handler on either an object method you provide, or on the SimGame delegate.

These are the types of Drawers SimGame offers:

- BlitDrawer – draws from a BlitSrc object (Mat2Xi/PaletteX or Mat2c)
- AnimBlitDrawer – uses a set of source mats; the animation cycle is controlled by the associated Cycler. A 2-blit animation would set phaseCount to 2; a 4-blit set to 4; etc.
- RadialAnimBlitDrawer – used a Mat of small blits that radiate outward according to the cycler; super useful for explosion-type effects. Invaders uses this for its exploding alien effect. The blits were just a 4x4 chop of a base alien blit rect.
- “Roll Your Own” – this is not a specific class, but is a Drawer created by using a TypedObject with a `handleRender` method, or else a method on your SimGame delegate object. Invaders uses this approach to handle its StarField entity’s drawing: it is just a `renderStars` method on the Invaders TypedObject delegate.

DrawerSet

A Drawer Set is a compound of a Drawer, a Cycler, and a MethodRef that can be used for manual drawing. Animated Drawers have their own drawing built in; the MethodRef is for “hand-rolled” Drawers, usually either a method on the SimGame delegate, or one on a TypedObject, if the Drawer needs additional state.

Putting it all together

Start small, and iterate your sim or game in steps. Don’t try to include every feature from the start. Begin with a subset of your game objects, and implement the code to render them. Then you can add some simple motion. This will motivate the need for boundaries, often using `stop`. It is quite rewarding to get to a point where things are

moving. Invaders started with the cannon, and the keyboard control. It was quite satisfying to see it rendered, and moving according to the key presses. The phase shot came next, and the obvious need to have it stop at the top boundary, otherwise it would literally go to “infinity and beyond!”.

Above all, have fun!

Misc

Small utilities and development aids that do not belong in a larger topic family.

They include:

- the useful `forget:` word for resetting the dictionary to before the word
- `Counter`, `Average`, and `MovingAverage` types for running totals/averages
- `Rand` and `Noise` for creating random numbers and Perlin noise (useful for things like terrain maps, where near neighbors want to be random but related.)
- `Stopwatch` for measuring elapsed time from a starting point

Reload Marks (forget:)

Interactive development often needs **load** → **try** → **wipe** → **reload** without relaunching the process. `forget:` defines an interpret-only **mark word**. Invoking that word erases the mark **and everything defined after it** in the current DictionaryContext (inclusive cut).

Mark-first script pattern

Put the mark **alone on line 1** of a reloadable script — no leading whitespace, no comment, nothing else on the line:

```
forget: reload
```

Session/core words stay outside the file (before the mark). Scratch definitions live under the mark.

`compileFromFile` reads that first line. If the named mark already exists, it runs it (wipe), **then** starts the compile-unit checkpoint and loads the file — `forget: reload` recreates the mark at the top. Load the same file again without typing the mark name first.

If line 1 is not exactly `forget: name`, `compileFromFile` does not auto-run anything (comments or `requiresFyne` before the mark disable the shortcut).

```
forget: reload

: helpers ...
TypedObject.newTypeDef^ ...
... makeType ...
: MyType#m ... ;

// load / reload:
"reloadable.fnl" compileFromFile
```

You can still type the mark name at the prompt to wipe without reloading.

Examples on disk often use `forget: reset` the same way (`examples/file-system/*.fnl` , format examples).

Behavior

Topic	Rule
Cut	Inclusive — mark word is removed too
Checkpoint	At mark create, stamps dict / CT / DT / local type-catalog high-waters
<code>compileFromFile</code>	Line 1 exactly <code>forget: name</code> → run mark if present, then compile-unit checkpoint
Invocation	Mark word is interpret-only (cannot compile its name into other words)
Scope	Current (task/session) DC — not sealed core

A failed reload still leaves the previous load wiped: the auto-run happens before the new unit's checkpoint.

Restriction: Final methods on surviving types

Before truncating, forget **refuses** (no mutation) if a type that **survives** the cut still has a Final method whose body was compiled **after** the mark (CT past the keep). That prevents dangling method nests.

Safe: create the type **and** its `: Type#... ; / : Type##... ;` methods **under** the mark (they roll back together).

Unsafe: register/ `makeType` a type **before** the mark, then add `: Type#m ;` **after** the mark — invoking the mark refuses and names the type/method.

Live MethodRefs after the cut

Forget drops types from the catalog; it does not free Go objects. A `MethodRef` (EventLoop handler, window `onClosed`, or a ref parked in a word defined *before* the mark) still points at the old type's method metadata.

On the cut, Final methods on types leaving the catalog are **poisoned**: invoking that `MethodRef` panics (`MethodRef: type Name#m forgotten`) instead of nesting into truncated or reissued CT. Close windows / drop outstanding refs before reload; the poison is the backstop if you do not.

What leave alone

Sealed / locked DCs, mid-compile, and core modules are out of scope. Foreign references held outside the DC are not tracked. Surviving defer targets that point into wiped dict/CT also refuse the cut.

Counter / Average / MovingAverage

Counter

`Counter` is a lighter-weight, non-threadsafe alternative to `AtomicInt`. It contains an `Int` value, with convenience methods such as `inc` `dec` etc. that do not return the new value. It is simpler to use than trying to juggle a value on a stack, fetch/increment/put back. It works especially well in a a local variable, eg

```
: countme
  Counter.new^ l.cnt!
  0 9 range
  foreach
    l.cnt.inc^
  /foreach
  "Count: " $ l.cnt@.value $ nl
;
```

Average

`Average` maintains a running average, variance, and N to-date without needing to retain the samples. It is mathematically correct, in the sense that the result is the same as accumulating the samples in a list then iterating an average/variance. Use `reset` to clear.

`Average` is threadsafe, and has `getMeanAndN` and `getMeanStdDevAndN` methods for getting the values atomically.

MovingAverage

`MovingAverage` is an exponential moving average (EMA) for signals that drift — live costs, rates, telemetry. Create a default with a roughly 20 item span with `MovingAverage.new^`. To create with a longer or shorter span, use `MovingAverage.newWithSpan^`; `.span` is fixed afterward (effective window in samples). Internally $\alpha = 2/(\text{span}+1)$. Same `.mean` / `.n` / `add` / `reset` / `getMeanAndN` API as `Average`, but no variance.

Prefer `Average` when you need an exact cumulative mean and variance; prefer `MovingAverage` when recent samples should outweigh old ones (e.g. SimClock update/render costs).

Random Values and Noise

Rand

`Rand` provides methods for creating pseudo-random values as follows. NOTE: do not use `Rand` for cryptographic contexts; use [SecureRandom](#) instead.

Method	Parameters	Purpose
<code>int</code>	<code>n:i – val:i</code>	Random integer from 0 to N-1
<code>float</code>	<code>(– val:f)</code>	Random float in $0.0 \leq \text{val} < 1.0$
<code>normFloat</code>	<code>(– val:f)</code>	Random float from normal distribution, mean=0 stddev=1

`Rand` has no instances — call class methods such as `Rand.int^`.

Noise

`Noise` generates Perlin noise: adjacent samples are random but related. Use it for terrain maps and other fields where nearby points should vary smoothly rather than independently (as with `Rand`).

Method	Parameters	Purpose
<code>new</code>	<code>(– n:Noise)</code>	Create a generator with a random seed
<code>newWithSeed</code>	<code>(seed:i – n:Noise)</code>	Create a generator with a fixed seed (reproducible)
<code>at1</code>	<code>(x:f – v:f)</code>	Sample at one coordinate
<code>at2</code>	<code>(x:f y:f – v:f)</code>	Sample at two coordinates (maps / terrain)
<code>at3</code>	<code>(x:f y:f z:f – v:f)</code>	Sample at three coordinates (e.g. treat z as time)

Sample values are typically near -1.0 ... 1.0.

```
Noise.new^ >ts
0.1 0.2 t:.at2^
// or reproducible:
42 Noise.newWithSeed^ >ts
0.1 0.2 t:.at2^
```

See also `examples/misc/noise-demo.fnl`.

Stopwatch

Stopwatch enables you to measure the time between a start time and a later time. You can use it in two modes:

- Stop mode – call the `stop` method when you want accumulation to stop
- Interval mode – call the `elapsed` method at a point you want the interval since `start`

Here is a typical use pattern:

```
: myword
  Stopwatch.new^ l.sw!
  //...
  l.sw.start^
  // intervening code
  l.sw.elapsed // ( int:Interval ) since .start
  // more code
  l.sw.stop^
  l.sw.elapsed // ( int:Interval ) since .start
  // use elapsed
;
```

There is no reset method; call `start` to begin a new timing session.

Final Modules and Projects

This release supports **local**, **leaf** modules and **.fnlp projects**. There is no remote repository client yet, and a module may not list other modules in `requires`. Behavior and file format may still change.

Example module: `modules/csv` (Table → CSV/TSV). Example project: `projects/csv-demo`.

Modules

A module is a directory with a `_MODULE_` JSON manifest (`name`, `version`, `final`, `files`, empty `requires`) plus the listed `.fnl` files.

install / import

```
// ( name:s opts:DO -- )
"github.com/androticus/final-go/modules/csv" newdo install

// ( syn:s name:s -- ) empty syn → default prefix (last path segment,
here csv)
"" "github.com/androticus/final-go/modules/csv" import
```

`install` compiles and seals the leaf module (already-installed is a nop). Resolution: optional `opts` field `path`, else a `_REPLACES_` map walked from the working directory, else `name` as a filesystem path.

`import` attaches an installed module to this Session. First attach wins. The default prefix is the last segment of the module name (`csv`). `local` and `core` are reserved.

Calling module words

Qualified: `prefix\word` (a backslash). After import, a **bare** name also searches attached modules.

```
csv\getDefaultCsvOptions
getDefaultCsvOptions      // same, via search order
```


using / using-default

```
// ( prefixes:StringList -- )
( "csv" s) using           // listed prefixes first in the search order
using-default              // ( -- ) restore Session.modules order
```

Security

`install` is denied by `SR.ModuleInstall`; `import` by `SR.ModuleImport`. A host that denies those bits should **pre-install and pre-import** anything the script is allowed to use. See [Security Restrictions](#).

Projects

A `.fnlp` file lists project `files` and `requires` (module names + versions). Version is carried but not used for selection yet.

```
// ( projectPath:s -- )
"projects/csv-demo/csv-demo.fnlp" loadProject
```

`loadProject` installs and imports each required module, then compiles the project files into the current Local Session. In **final, File → Load** on a `.fnlp` does the same.

`loadProject` itself is not SR-tagged; the nested `install` / `import` still honor the Session mask.

Application Services

Final provides basic application services often needed in embedding scenarios, or when building an application using Final. Most of these services present a single usage API in Final, but can be configured in two complementary but compatible ways:

1. By the embedder, when initializing a Session;
2. In Final, during Session startup.

Final provides the following Application Services, as of this release:

- [Configuration](#): a configuration object built from a JSON string, accessible as `Session.config` via `sess.config` in Final code,
- [Logging](#): a Logging interface, with a pluggable `LogProvider`, accessible as `Session.logger` via `sess.logger` in Final code
- [Cmdline Scripting and OS Piping](#)

Application Configuration

Final configuration is based on JSON.

Final provides Session configuration via a `DynamicObject`, that can be initialized with JSON.

Final API

```
DynamicObject Session.config via sess.config
```

Embedder Initialization Options

Once the Session has been created, the following call can be made to set `Session.config` from a JSON string. Note that this call invokes a `Send` on the VM, in order to execute Final code. JSON parsing and rendering in Final is all done with Final code, there is no native code involved.

```
sess := NewSession | NewLockSession

err := sess.InitConfig(jsonString)
```

Final Initialization Options

Session provides an api for initializing `sess.config` with a JSON string. The api is set up this way so it works for both Embedder and Final initialization:

```
// 1. put your JSON on session.configJson
`{ "paramA" : 1, "paramB" : 2 }` sess.configJson!

// 2. Call the init
sess.initConfig^
```

This replaces the default empty `sess.config` object with one initialized from the JSON string supplied. This call should be made during your application startup, before any code tries to access the config members. Successful initialization also clears `sess.configJson`

Using sess.config

Session.config via sess.config is a standard DynamicObject. It will take the shape of the JSON string that was used to initialize it. The various properties of the config can be accessed using standard compound object syntax. For example, using the schema from the initialization done above:

```
: print-config
  ( sess.config.paramA@ sess.config.paramB@ )
  "paramA: %d paramB: %d\n" fmt $
;
print-config

paramA: 1 paramB: 2
```

You can also store nested fields on the live config object without parking it:

```
3 sess.config.paramA!
```

Logging

Final provides a `Session.Logger` object, and helper words, that provide the application logging interface. Logging is based on levels of severity, with the logger operating at a given level. Logs targeted for the active Level or lower get logged, other logs get dropped. A conditional word to test for each level is provided, to enable conditional preparation of logging messages that might entail formatting costs.

The underlying logging mechanism is supplied by a `LogProvider` object. This can be supplied by an embedder, during Session creation, or in Final, on Session initialization. Final provides 2 types of default `LogProviders`: a simple File-based logger, based on a `TextWriter`, and a `StdOut` or `StdErr` logger.

Logging Levels

The following Levels are provided, and must be supported by any `LogProvider`.

```
Levels:
0 Off
1 Fatal
2 Error
3 Warn
4 Info
5 Debug
6 Trace
```

Session.Logger

`Session` holds an object of type `Logger` via `sess.logger` that is the logger shared by all Pips in the Session. It has these principle members (use `help Logger` for full description):

```
Logger
  Int level readonly const property
  String service property
  String instanceId property

  // ( msg:s data:a level:i -- )
  log
```

The `service`, optional `instanceId` and `level` are typically set once during startup. Here are the logging levels

Final Logging Words

In your final code, you will typically use the following words to log. These abstract all the details, and enable concise logging logs in your code.

Even if you send a log, it will only get written to the underlying Provider if the level of the log is less than or equal to the current `Logger.level`, and logging is enabled (level `<= 0`). It is typical for an application to operate at Info level, or at a minimum, Warn. Conditional test words for all log levels are provided, that can be used to put your log statment in an `if / then` if preparation of the message or data will entail a significant runtime cost. For most eg Info or below messages, with a simple unformatted string message, and no data, the extra cost of the test is usually not worth the complexity.

All the words below delegate to a single word `logAt` which also includes the level at which to log. You will typically never use `logAt`, and will use one of these, or the test then then the log word:

```
Words:
// logs, only if level <= Logger.level, and level <= 0
// ( msg:s data:a level:i -- )
logAt

// ( -- b ) ( msg:s data:a -- )
logFatal?   logFatal
logError?   logError
logWarn?    logWarn
logInfo?    logInfo
logDebug?   logDebug
logTrace?   logTrace
```

Typical logging scenarios, simple, and with JSON data:

```
// simple log, no data
"Something simple happened" "" logInfo

// providing an existing object as data context
"Something else happened" myObj >json logInfo

// a log, with constructed JSON data
"Something happened" { 1 "param1" 2 "param2" } >json logInfo
```

Note: use the compact `>json` operator, not the pretty-print one `>jsonp` to avoid spurious newlines and unnecessary whitespace.

Conditional logging:

```
// costly/elaborate Trace message formatting
logTrace? if makeTraceMessage makeTraceData logTrace then
```

Logger Initialization

In Final

Initializing logging takes 2 simple steps:

1. Obtain a LogProvider, from a LogProvider factory method;
2. Call logger.init with your service/instanceId/level and the provider.

See the example code `examples/app-svcs/logging.fnl` for details.

By an Embedder

Implement a LogProvider that conforms to the following exported interface.

```
type LogProvider interface {
    Log(service, instanceId string, level int, msg string, data any)
}
```

The Log method will only be called if the LogProvider is enabled, and the `level` is at or below the current LogProvider level. The LogProvider must be threadsafe, and must serialize logs. It should not block in normal operation; blocking under unusual conditions of backpressure by the underlying log writing mechanism is acceptable. Logs should not be dropped.

The `data` parameter will typically contain:

- an empty string; for database logging, this can be treated as NULL
- a string, which can be assumed to be valid, compact JSON
- a Go `nil` interface value; can be treated as NULL or empty

The LogProvider may apply additional interpretations of another type of value in data, for its own native log rendering. For example, it can check for map or [] and call `json.Marshall` on the value.

Final for command line scripts and OS Pipes

Final supports command line operation to run scripts written in Final, or to be the target of a pipe. It uses semantics similar to Python and node.

All of the below assume you have put the Final folder on your path. Otherwise, use `./final`.

To launch `final` and run a script, without launching the interactive tui, use this format:

```
final commandScript.fnl [ param1 param2 ...]
```

To pipe text into Final's stdin, do the following:

```
someShellCmd | final commandScript.fnl [ param1 param2 ... ]
```

When you actually want to pipe in Final interpretable text, do the following. The `-` is only needed if you add params to the command line – it is a placeholder for the stdin input.

```
echo '1 2 + $ nl' | final - [ param1 param2 ... ]
>3
```

Final also support creating executable final scripts on Mac/Linux, using shebang invocation. (Note: as with all executable scripts, you must mark it executable using `chmod`.) `final` must be on the PATH.

eg

```
#!/usr/bin/env final
sys.argv >json $
```

In all cases, the command and any parameters are available in Final via the `Sys.params` StringList object. `sys` instance is available via the axiom `sys`. `params[0]` contains the command, or `-` `params[1..]` contain any params you passed.

```
// this assumes there was a command. otherwise params is empty
0 sys.params.@() $
```


If you need to access the raw, unprocessed command line arguments, C-style, they are available in `sess.argv` – `[0]` is the text that launched Final, eg `./final`, then the remaining values are the individual arguments. Note: you should generally not need to access this.

AI Tool Guidelines

Final is designed to be a **tool an agent can call**: the host creates a Session, the model emits a `script` string, and Final returns captured stdout. The sample is `cmd/final-ai` (web chat; `-dev` adds a Final Dev console that shares the Session).

Details of Session / Send live in [Embedding Final](#). This page is the short policy for that use.

Sandbox first

Give the model a **restricted** Session. Set [Security Restrictions](#) so untrusted scripts cannot open files, dial HTTP, listen, spawn Pips, open a database, use crypto, or `install` / `import` modules unless you intend that.

```
sess, err := finalgo.NewLockSession()
// ...
sess.SetSecurityRestrictions(
    finalgo.SR_FileRead | finalgo.SR_FileWrite | finalgo.SR_FileSystemRead
|
    finalgo.SR_HttpDial | finalgo.SR_HttpListen | finalgo.SR_Threads |
    finalgo.SR_Db | finalgo.SR_Crypto |
    finalgo.SR_ModuleInstall | finalgo.SR_ModuleImport,
)
```

Prefer **LockSession** for agent turns: `GetLockedSender()`, run the tool `Send(s)`, then `Unlock()`. That keeps a Dev REPL or a second HTTP handler from interleaving dictionary changes mid-turn.

Tool shape

`finalgo.NewFinalTool()` is a static OpenAI-style tool named `final`. `Dispatch` parses `{"script":"..."}` and calls `Sender.Send`. Register `Name` / `Description` / `ParametersJSON` with your provider; pass the **sender handle**, not the Session, into `Dispatch`.

Keep the tool description honest about what the Session can do. If you denied files and HTTP, do not advertise “read this path” or “fetch that URL.”

Scripts the model writes

- Prefer short postfix `$` results over dumping huge structures.

- Invalid JSON and similar input faults **throw**. An uncaught throw looks like a panic to the host — wrap parse of model- or user-supplied text in `try / catch` inside the script, or treat `Send` errors as the tool result.
- Do not rely on values left on stacks between tool calls unless you created a **persistent** Session. The default embed Session clears stacks each `Send`; **words** persist.

Related

- [2 Embedding Models](#) — `GetSender / GetLockedSender, FinalTool`
- [Security Restrictions](#) — SR catalog including `ModuleInstall / ModuleImport`
- [Error and Exception Handling](#) — `throw` vs panic

How Final Works

Final is written in Go, and takes full advantage of all the features Go offers:

- fully self-contained binaries
- Garbage Collected memory management
- Memory allocator optimized for modern workloads
- rich set of building blocks, such as Utf-8 strings, growable slices (arrays), maps
- generics, to simplify creating a template class and typed instantiations
- excellent threading model in goroutines and channels
- simple project setup, and inclusion of packages
- fast build times

Final itself was designed to be highly efficient at runtime, both in terms of memory consumption, as well as execution speed.

Final Virtual Machine (VM)

Final code executes in a Virtual Machine. Final code is a combination of `Axioms`, which are written in Go, Go-native Type member accessors written in Go, and Final Colon Words.

An Axiom is a fundamental operation, such as `+` `dup` and other such low-level operations that provide the core “machine code” of Final.

OpCodes and the Code Table

Each Final Axiom has an integer `OpCode`, that gets compiled into something called the Code Table, which is a Go slice of ints. The VM has an Instruction Pointer (IP) that points to the next cell in the CT. The basic operation of the VM is to read the next OpCode, then dispatch to its Go code, using an efficient Go switch statement that is compiled at the machine code level to a fast, efficient jump table. Due to how modern processors cache often-used instructions and data, the Final VM is blazing fast, and overall, often comparable to compiled machine code as generated by languages like Go. Because the actual machine code making up Final programs is more limited than large programs in compiled languages, most of its actual machine code typically stays “hot” in the L0 or L1 cache of the CPU, resulting in excellent performance. So don’t imagine that because Final compiles to a VM, that its code will be substantially slower than native code!

The other place Go-native code appears, is in so-called native members of most of Final's runtime types. But again there, it overall represents a relatively small amount of compiled machine code, and also benefits from effective hot caching.

The Code Table holds the following types of entries:

- Axiom OpCodes
- so-called “arming” prefixes, that provide a needed subcode, context selector, or Data Table index (see below) to the following OpCode
- literal Bool, Int, Uint or Float values, read by a DoLiteral opcode, that reads the immediate value, and pushes the literal on DS
- Nest, followed by the address of a compiled colon word; this pushes the current IP on the Return Stack (RS), and dispatches the VM to the code of the colon word
- Unnest, which is compiled by `;` that returns to the code after the colon jump

The Data Table

To keep execution efficient, the Code Table is limited to a single int value in each slot. When a literal that takes more space needs to be compiled for a word, such as a String literal, the word allocates a slot in the Data Table, which is a go slice of ‘any’ values, which are 128 bits long. Data Table slots are also used to hold constants and variables defined in Final source, and for some other specialized purposes.

Dictionary Context

The trio of a Dictionary, Code Table, and Data Table make up what is called a `Dictionary Context` (DC). A compiled colon word has all its parts colocated in a single DC. During bootstrap and compiling of core Final words, this DC is the global core DC. When you compile your words, you are compiling into your Session-local DC. Colon words in one DC can seamlessly include words in another DC; the code needed to jump and return to that DC's Code Table is compiled automatically into the Nest operation.

Stacks

All of Final's stacks are instances of a Go Stack class, which is built on a Go slice of `any`. The stacks are checked at runtime for underflow, and overflow, to limit the effects of runaway code or unchecked recursion. The Data Stack is given a small initial capacity value, and the auxilliary stacks none. All stacks automatically expand as needed, up to their limit.

Tasks and Sessions

From a user perspective, Final code executes under the auspices of a Task, in something called a Session. The Task hold static information about the runtime context, such as its Dictionary Context. The runtime is fully multi-threaded, and supports any number of Task/Session pairs.

A Session provides the API for communicating with the the instance of the VM defined by the task. It includes a Send function that receives text, provides it as input to the running instance, and waits for a reply. This is the basis for the final Prompt/Reply sequence, and the embedded instance Send command, for example an AI Tool prompt, that can include word definitions and immediate commands.

Error Handling

Each running instance of the Final VM captures runtime errors, and resets the VM to a known good state. It reports the fault, resets the stacks, and does a “warm boot” of the VM, which puts in back into interpret mode. Final also includes Structure Exception Handling, described in the section [Error and Exception Handling](#) .

Threading

Pips are based on goroutines, so enjoy the benefits of an M:N dispatch model that limits the number of actual OS threads to what the CPU can provide via cores, then using an efficient dispatch routine to assign ready-to-run goroutines/Pips to an available core.

Pipes are based on go channels, and likewise enjoy all their benefits, such as being threadsafe, efficient, and (fairly) easy to reason about.

Each Pip in a Session gets a clone of its parent Task, and shares the Dictionary Context with it. Each Pip gets its own set of stacks, and executes in its own instance of the VM run loop. When the spawn word of a Pip finishes, it returns to an Exit opcode, which causes the run loop to stop, and the underlying goroutine to finish. It reports that it is done via a control channel, to its parent. The DC of the root Task remains locked as long as any child Pips exist. Once all end, the DC unlocks. It would not be unusual for a Final program to compile all its words, create any global variable or constants need, and then spawn its workers, rendering the DC locked for the remainder of the runtime.

Because spawn: requires a definite Final colon word to be dispatched, Pips do not have an interpreter, they simply execute the code of the spawn word.

By default, spawned words inherit the stdin/out of the parent. The stdio streams are threadsafe, and can thus demultiplex the output from child Pips, if writing to stdout is a component of the design.

Files and I/O

Final's Stream classes are based on a 2-level division of surface and provider. The surface level provides the APIs shared by all particular types, such as ReadByte or WriteLine; provider objects for each type of Stream, such as file, stdio, string, etc. are plugged in to the surface object, and the surface object simply delegates to the provider. This design provides good performance, simplicity, and extensibility, for example to compose streams that have intermediaries that do things like compress, buffer, or encrypt/decrypt.

Embedding Final in Go

Final was designed first as a language for **AI tools**, and second as a general-purpose stack language you can drive from Go. The [Introduction](#) and [Features](#) survey already mentioned embedding; this section is the **host API** guide.

You embed Final by creating a **Session**, then calling **Send** with a string of Final source. Each **Send** is one interpret/compile batch: the script runs to completion (or faults), captured stdout comes back as a string, and dictionary state in that Session persists for later Sends.

Page	Contents
2 Embedding Models	<code>finalgo</code> (simple) vs <code>finalembd</code> (host types); Session / Send / async; <code>LockSession</code> ; agent sample app
Security Restrictions	Disabled-set capability model for sandboxed AI and other hosts

Already covered elsewhere

- Interactive use without embedding — [The Final TUI](#)
- File / HTTP / threading affordances that SRs can deny — [File Streams](#), [Web Dialing](#), [Threading](#)
- How a Session-local dictionary relates to the VM — [How Final Works](#)

Which package?

- Prefer `finalgo` for agentic and request/response hosts: create a Session, `Send` scripts, optionally expose `FinalTool` to an LLM. That is the path used by the sample chat app.
- Use `finalembd` when the host must register its own Go types so Final scripts can construct and drive them (widgets, domain objects, generators).

Details and tradeoffs: [2 Embedding Models](#).

2 Embedding Models

Final exposes two public Go packages for embedding. Both boot the runtime on import, own a long-lived **Session**, and run scripts with **Send**. They differ in how much of the host you expose into Final.

Package	Role	Choose when
finalgo	Minimal Session / Send / Close API; optional FinalTool for LLM tool calling	AI agents, chat backends, sandboxed script runners, most “run this Final string” hosts
finalembd	Same Session/Send model plus host type registration (RegisterTypes) and re-exported Final cell/collection types	Embedding Final as a scripting language for <i>your</i> Go types (UI, domain objects, generators)

For **AI agentic support, prefer the simple model (**finalgo**)**. Agents need a stable interpret tool, captured stdout, and a restricted Session — not a catalog of host widgets. Reach for **finalembd** only when scripts must mint and drive types that live in your application.

Concepts: Session, Send, and async

Session

A **Session** is a long-lived Final execution context:

- Private **module dictionary** (Session-local DC) where colon words you define with **:** persist across Sends
- One dedicated **VM worker** goroutine that runs script batches
- Optional **Security Restrictions** disabled-set (see [Security Restrictions](#))

Create one Session per logical agent conversation, web client, or embed unit. Close it when that unit ends.

Send

```
out, err := sess.GetSender().Send(script, timeouts)
```

Argument / result	Meaning
<code>script</code>	Final source (interpret and/or compile). Empty script is an error.
<code>timeouts</code>	Advisory run budget in whole seconds (<code>0</code> = none). Used for inward I/O defaults; not a hard wall-clock kill of blocking natives.
<code>out</code>	Captured stdout text from the batch (<code>\$</code> , <code>.s</code> , stream writes to stdout, ...)
<code>err</code>	Fault or interpret error; may be accompanied by partial <code>out</code>

Default Sessions use **ephemeral stacks**: each `Send` starts with empty DS/RS/... Dictionary and compiled words **persist**. That matches request/batch and agent-tool semantics. Persistent stacks (REPL-like) exist for the TUI path (`finalembd.NewPersistentSession` / `finalgo.NewREPLSession`) — not the usual agent default.

`Send` **blocks** until the worker finishes the batch. Concurrent `Send` calls on the same Session **serialize**.

Call `sess.SetSecurityRestrictions(mask)` as a Session-wide floor, or `sender.SetSecurityRestrictions(mask)` on the handle from `GetSender()` before that `Send` (those bits are **ORed** with the floor — you can tighten, not lift). Bits **set** deny the matching [SR](#) themes.

Making Send asynchronous

The VM already runs on a **worker goroutine** inside the Session. Your host call to `Send` is the synchronous rendezvous with that worker.

To keep an HTTP handler, chat turn, or UI thread responsive, run `Send` on **your own** goroutine (or job queue) and deliver `out` / `err` when ready:

```

go func() {
    out, err := sess.GetSender().Send(script, 30)
    // deliver out/err on your completion channel, websocket, etc.
    _ = out
    _ = err
}()

```

Do not start multiple overlapping `Send`s and expect parallel VM progress on one Session — they queue. For cancel, call `sess.Break()`; the VM stops at the next opcode boundary (natives may still finish first).

LockSession (exclusive agent turns)

`finalgo.NewLockSession()` / `finalembed.NewLockSession()` returns a `LockSession`. Its only Send path is `GetLockedSender()`, which holds until that handle's `Unlock()`. A second `GetLockedSender` **blocks** until `Unlock`. `final-ai` uses this so a Dev console cannot interleave Sends in the middle of an AI turn.

You may `Send` several times on the same locked sender (compile helpers, then a follow-up) and then `Unlock`. After `Unlock` the handle is invalid (`ErrLockedSenderInvalid`).

Simple model: finalgo

Import path: `github.com/androticus/final-go/finalgo`.

Core API

API	Effect
<code>NewSession()</code> (<code>*Session</code> , <code>error</code>)	Local module Session; ephemeral stacks; unrestricted by default
<code>NewLockSession()</code> (<code>*LockSession</code> , <code>error</code>)	Same, but Send only via <code>GetLockedSender</code>
<code>(*Session) GetSender()</code> <code>*SessionSender</code>	Handle for one or more <code>Send</code> s; optional <code>SetSecurityRestrictions</code>
<code>(*LockSession) GetLockedSender()</code> <code>*LockedSender</code>	Exclusive hold until <code>unlock()</code>
<code>(*SessionSender) Send(script string, timeoutS int) (string,</code>	Run one batch; return stdout

API	Effect
<code>error)</code>	
<code>(*Session) Break()</code>	Cooperative stop of in-flight VM run
<code>(*Session) Close() error</code>	Stop worker; idempotent
<code>NewFinalTool() *FinalTool</code>	OpenAI-compatible tool wrapper (<code>Name</code> / <code>Description</code> / <code>ParametersJSON</code> / <code>Dispatch</code>)
<code>Sender</code> interface	<code>SetSecurityRestrictions</code> + <code>Send</code> — what <code>FinalTool.Dispatch</code> uses

Minimal embed (agent-friendly)

```
package main

import (
    "fmt"
    "log"

    "github.com/androticus/final-go/finalgo"
)

func main() {
    sess, err := finalgo.NewSession()
    if err != nil {
        log.Fatal(err)
    }
    defer sess.Close()

    snd := sess.GetSender()

    // Interpret a postfix expression; $ prints the TOS number to stdout.
    out, err := snd.Send(`3 4 + $`, 0)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Print(out) // 7

    // Definitions persist for later Sends in this Session.
    if _, err := snd.Send(`: sq dup * ;`, 0); err != nil {
        log.Fatal(err)
    }
    out, err = snd.Send(`5 sq $`, 0)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Print(out) // 25
}
```

That is the whole loop for most agentic hosts: **NewSession** → **Send** → **Close**.

Final as an LLM tool

`FinalTool` packages the Session for providers that accept OpenAI-style tools. `Dispatch` parses `{"script":"..."}`, calls `Sender.Send`, and returns captured stdout (or an `[error] ... content string`).

```

sess, err := finalgo.NewSession()
if err != nil {
    log.Fatal(err)
}
defer sess.Close()

ft := finalgo.NewFinalTool()
// Register ft.Name(), ft.Description(), ft.ParametersJSON() with your AI
client.
// On tool_calls:
content, err := ft.Dispatch(sess.GetSender(), toolCallID, ft.Name(),
argumentsJSON)

```

Sample chat application

This repo ships `cmd/final-ai`: a small web chat UI that registers Final as a tool the model can call. Early tests showed the agent successfully calling Final for **mathematical calculations** (postfix scripts via the `final` tool). Run it with your provider API key (see `cmd/final-ai`); use `-dev` for the tabbed AI Chat + Final Dev console that shares one Session per browser cookie.

Extensive model: `finalembed`

Import path: `github.com/androticus/final-go/finalembed`.

Use this when Final scripts must create and operate on **your** Go types. You register each host type once after `NewSession`, before any `Send`. Registration is app-global for that Session's module.

Core API (beyond `finalgo`)

API	Effect
<code>NewSession() (*Session, error)</code>	Local module; ephemeral stacks
<code>NewLockSession() (*LockSession, error)</code>	Exclusive Send via <code>GetLockedSender</code>
<code>NewPersistentSession() (*Session, error)</code>	Same, but stacks survive across Sends (REPL / TUI style)
<code>(*Session) RegisterTypes([]TypeReg) error</code>	Install host types; call once before Send

API	Effect
<code>NewTypeReg[T](name string, reg func(*Type)) TypeReg</code>	Build a registration entry for Go type <code>T</code>
<code>(*Type) RegisterProperty / RegisterNativeMethod / RegisterNativeClassMethod / ...</code>	Describe members on the Final type
Re-exports in <code>types.go</code> / <code>cells.go</code>	<code>Int</code> , <code>Float</code> , <code>String</code> , <code>AnyList</code> , <code>ToCell</code> , ... for DS-safe boundaries

Host type sketch

```

sess, err := finalembed.NewSession()
if err != nil {
    return err
}
defer sess.Close()

err = sess.RegisterTypes([]finalembed.TypeReg{
    finalembed.NewTypeReg[*Widget]("Widget", func(t *finalembed.Type) {
        t.RegisterNativeClassMethod("new", func(f *finalembed.RunFrame,
ths *finalembed.TypeObjPair) {
            f.RP.DataStack.Push(NewWidget())
        })
        t.RegisterProperty("title", "String",
            func(inst any) any { return finalembed.String(inst.
(*Widget).Title) },
            func(inst any, v any) { inst.(*Widget).Title = string(v.
(finalembed.String)) },
            false,
        )
        t.RegisterNativeMethod("render", func(f *finalembed.RunFrame, ths
*finalembed.TypeObjPair) {
            ths.Obj.(*Widget).Render()
        })
    }),
})
if err != nil {
    return err
}

_, err = sess.GetSender().Send(`Widget.new^ >ts "Hi" t:.title! t:.render^
t:drop`, 0)

```

Cell rule: values that cross the data stack (property get/set, method args/results) must be Final cells — `finalembed.Int`, `Float`, `String`, ... — or pass through `finalembed.ToCell`. Raw Go `int32` / `float32` will eventually panic under arithmetic.

A fuller worked example lives in `cmd/lissajous` (register a generator type, drive it from a Final script).

Tradeoffs vs `finalgo`

Concern	<code>finalgo</code>	<code>finalembed</code>
Lines to first useful <code>Send</code>	Very few	Registration boilerplate per host type
API for an LLM tool	Excellent (<code>FinalTool</code>)	Possible, but usually unnecessary
Script your own Go structs	No	Yes
Re-use Final Lists/Maps from Go natives	Limited	First-class via re-exported types
Typical security posture	Tight SR mask; math / pure logic	Often needs the affordances your types wrap (files, UI, ...)

Recommendation: start with `finalgo` for agents. Graduate to `finalembed` only when a script must say `YourType.new^` and call methods on host objects.

Related

- [Security Restrictions](#) — deny file I/O, HTTP dial, threading for sandboxed Sessions
- [Features](#) — embedding, `LockSession`, and SRs listed in the survey
- [AI Tool Guidelines](#) — sandboxing and exclusive turns for LLM tools
- Design depth (not required for host authors): `docs/003-design-notes/134-embedding-final.md`

Security Restrictions

Security Restrictions (SR) are themed capability groups a host can **disable** for a Final Session. When a theme bit is set in the Session's disabled-set mask, tagged system words and type members that require that capability fail with a fixed `"operation not allowed"` — at compile time, interpret time, dictionary lookup, and as a safety net on native / compound invoke.

Policy is enforced in **Go at language boundaries**, not inferred from prompts. That makes SRs suitable for giving an AI (or any untrusted script source) access to Final **without** opening the filesystem, the network, or threading.

File factories, HTTP dialing, and pip spawn already participate in this model; see also [File Streams](#), [Web Dialing](#), and [Threading](#).

Model

Idea	Meaning
Disabled-set mask	Integer bitfield on the Session / Task. Bits set = themes denied.
Tag on affordance	Privileged words and type members carry SR bits (e.g. file-open factories → <code>FileRead</code> / <code>FileWrite</code>).
Violation	<code>(taggedSR & Task.restrict) != 0 → deny</code>
Unrestricted	Mask <code>0</code> — full vocabulary (normal TUI / trusted host default)

Axioms such as arithmetic, stack ops, and `str::` / `dt::` families are **not** SR-gated. Denial starts at **system colon words** and **Type members** that are privileged affordances (open file, dial HTTP, spawn a pip, ...).

On the Final side, theme constants live on the synthetic type `SR`:

```

SR.ModuleInstall
SR.ModuleImport
SR.FileRead
SR.FileWrite
SR.FileSystemRead
SR.HttpDial
SR.HttpListen
SR.Threads
SR.Db
SR.Crypto
SR.Mask

```

Active catalog

SR	Disallows
ModuleInstall	The <code>install</code> axiom (and nested <code>install from loadProject</code>)
ModuleImport	The <code>import</code> axiom (and nested <code>import from loadProject</code>); pre-import in the host if scripts must still see a module
FileRead	Open-for-read / content stream factories from path strings
FileWrite	Open-for-write / create / append factories (includes directory creation)
FileSystemRead	Enumerate / metadata / exists-style filesystem queries (not the same as opening content)
HttpDial	Outbound HTTP client ops (<code>HttpDialer.send</code> , <code>httpDial</code> , ...)
HttpListen	Creating an HTTP listener
Threads	Creating Pips / Pipes, <code>spawn</code> : / (<code>spawn</code>)
Db	Opening a database via <code>DbProvider.new</code>
Crypto	Digests (<code>Hash.hash</code>), Hex/Base64 encode/decode; later AEAD / sign / KDF / SecureRandom

Combine themes with bitwise OR. A typical **AI tool Session** denies file / network / threads (and often **Db** / **Crypto**) so the agent can define words and run math / data logic, but cannot touch disk, network, or crypto primitives. Leave **Crypto** enabled if the agent needs content-id digests.

Host usage

Session default

The disabled-set is fixed when the Session is created (copied onto the Task). Every normal `Send` runs under that mask.

- Interactive CLI: `final-go -restrict <mask>` (hex-friendly), e.g. deny threads with `-restrict 0x2000000000000000`
- Embedders: `sess.SetSecurityRestrictions(mask)` after `NewSession` / `NewLockSession`, or `sender.SetSecurityRestrictions(mask)` on the handle from `GetSender()` / `GetLockedSender()` before `Send`.
`finalgo.NewREPLSession(module, restrict)` still takes the mask at construction. `NewSession()` defaults to unrestricted (`0`). Prefer a restricted Session for any agent-facing tool.

Agent-shaped example

```

package main

import (
    "fmt"
    "log"

    "github.com/androticus/final-go/finalgo"
)

// Disabled-set: bits set = denied. Matches SR.FileRead | FileWrite |
// FileSystemRead |
// HttpDial | HttpListen | Threads (see catalog above).
const agentRestrict = int64(
    0x1000000000000 | // FileRead
    0x2000000000000 | // FileWrite
    0x4000000000000 | // FileSystemRead
    0x2000000000000 | // HttpDial
    0x4000000000000 | // HttpListen
    0x2000000000000, // Threads
)

func main() {
    // Persistent stacks are fine for a long-lived tool Session;
    // dictionary state
    // is what agents usually need across tool calls.
    sess, err := finalgo.NewREPLSession(finalgo.REPLModuleLocal,
    agentRestrict)
    if err != nil {
        log.Fatal(err)
    }
    defer sess.Close()

    snd := sess.GetSender()
    out, err := snd.Send(`3 4 + $`, 0)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Print(out) // 7 – pure compute still works

    // File / HTTP / spawn affordances under this mask fail with
    // "operation not allowed" instead of performing the side effect.
    _, err = snd.Send(`"secret.txt" TextReader.newFromFile?^`, 0)
    if err == nil {
        log.Fatal("expected restriction denial")
    }
    fmt.Println("denied:", err)
}

```

With that mask, an LLM tool backed by `finalgo.NewFinalTool()` can still run postfix math, build Lists/Maps, parse Json in memory, and compile helper words — without enabling file ops, web dialing, or threading.

For the shortest unrestricted agent path, `finalgo.NewSession()` remains fine in trusted environments; add a restrict mask whenever the script source is model- or user-controlled.

Per-Send extra restrictions

`sender.SetSecurityRestrictions(mask)` on the handle from `GetSender()` / `GetLockedSender()` **ORs** those bits with the Session floor for that `Send`. You can tighten a dispatch; you cannot lift a Session-level denial.

What SRs are not

- **Not** a substitute for OS sandboxing, containerization, or network policy outside the process
 - **Not** instruction-budget or hard wall-clock kill (those are separate runtime options; `timeouts` on `Send` is advisory)
 - **Not** path allowlists — host path roots, when added, complement SR bits rather than replace them
-

Related

- [2 Embedding Models](#) — Session / Send and why agents prefer `finalgo`
- [Features](#) — SRs listed in the survey
- Normative design detail: `docs/003-design-notes/143-security-restrictions.md`

VS Code and Cursor — Final extension

Final ships a **language extension** for **VS Code** and **Cursor** (same VSIX; no Cursor-specific fork). It registers language id `final` for `*.fnl` files and provides TextMate highlighting, 2-space indent defaults, comment toggle, snippets for sandwiches, and same-line `"` auto-close.

Package home in the repo: `extensions/vsc/` (extension id **final-lang**, version **0.0.7**).

Install is **local** on your machines (F5 Extension Development Host, or Install from VSIX). It is not published to the Marketplace. No Final runtime binary is required for this pack — it is declarative editor config plus grammar and snippets.

Features

Area	What you get
Highlighting	<code>//</code> and <code>/* */</code> comments; <code>"..."</code> (escaped) and <code>`...`</code> (raw) strings; <code>'...'</code> runes; definition name after <code>:</code>
Indent	2-space defaults for <code>[final]</code> ; Tab / Shift-Tab; Enter indent after sandwich / colon openers; outdent on closers
Line endings	LF (<code>\n</code>) preferred for <code>.fnl</code>
Comment toggle	<code>Cmd+/</code> / <code>Ctrl+/</code>
Snippets	colon-def, control-flow sandwiches, flush-left block comment and raw string (prefix + Tab)
"..." pair	typing <code>"</code> inserts the closing quote on the same line

Not in this pack

- No coloring of ordinary words as “keywords” (Final has no special keyword class for axioms)
 - No rune / unicode-character autocomplete (type `'...'` yourself; it still highlights)
 - No language server yet — no dictionary completion, stack-effect hover, go-to-definition, or diagnostics
-

Install

From the Final tree (recommended while developing)

1. Open `extensions/vsc` as the workspace folder in VS Code or Cursor.
2. Run **Run Extension** from the Run and Debug view (**F5**) — opens an Extension Development Host with Final loaded.
3. Or package and install a VSIX:

```
cd extensions/vsc
npx --yes @vscode/vsce package --allow-missing-repository
```

Then in VS Code or Cursor: **Extensions** → ... → **Install from VSIX...** → choose `final-lang-0.0.7.vsix` (or the `final-*.vsix` just built). A packaged VSIX may already be present under `extensions/vsc/`.

After install

Open any `.fnl` file — language mode should be **Final**.

Requirements: VS Code or Cursor meeting `engines.vscode` in `extensions/vsc/package.json` (currently `^1.85.0`).

Snippets

Type the prefix and press Tab:

Prefix	Inserts
<code>def</code> or <code>:</code>	<code>: name / indented body / ;</code>
<code>if</code> / <code>nif</code>	sandwich closed by <code>then</code>
<code>switch</code>	<code>... /switch</code>
<code>foreach</code>	<code>... /foreach</code>
<code>forever</code>	<code>... /forever</code>
<code>begin</code>	<code>... closer choice (again / until / repeat)</code>
<code>/*</code>	block comment, flush left
<code>raw</code> or <code>`</code>	raw string, flush left

Colon-def and sandwiches indent the body by 2 spaces. Block comments and raw strings stay at column 0.

Colors (theme scopes)

Scopes are theme-driven. Typical scopes:

Form	Scope
//	comment.line.double-slash.final
/* */	comment.block.final
"..."	string.quoted.double.final
`...`	string.quoted.other.raw.final
'...'	constant.character.final
name after :	entity.name.function.final

Notes

- Indent is cosmetic — the Final parser does not care about indentation.
- Prefer associating `.fnl` with this extension (or your editor of choice) when working through [Installation](#) and [Hello Final!](#).
- Extension README in-tree: `extensions/vsc/README.md`.

Raspberry Pi 5 Support

This release of Final includes preliminary support for controlling a Raspberry Pi 5 using Final. It has been tested on a full Raspberry Pi 5, but may also work on a Pi Zero 2 running 64-bit Pi OS. The Final runtime itself is under 16 MB, so it adds little to the distribution size. Final has no library dependencies or framework; it is entirely self-contained.

Support in this release is **GPIO line in/out** (BCM/SoC numbers). Treat it as experimental: kit case LEDs and buttons are often owned by the OS.

Final can run **headless** as a controller. It loads a platform manifest, then an optional user `.fnlm` / `.fnl` (`-m`). Pi packs load `extras/platform/linux/pi/gpio.fnl` automatically (`gpio0 ... gpio27`, `board>bcm` / `bcm>board`).

Numbers are **BCM**, not header pin numbers (`gpio17`, not header 11):

```
gpio17 Pin.out^ var: led
led @ .high^
led @ .low^

gpio27 Pin.in^ var: btn    // pull-up; active-low typical
btn @ .isLow?^

11 board>bcm                // → 17
```

Optional `FINAL_GPIOCHIP=...` (some Pi 5 kernels use `gpiochip4`). The process typically needs `gpio` group access to `/dev/gpiochip*`. Use `help Pin` on a Pi pack for members (`.high^` `.low^` `.toggle^` `.isHigh?^` `.isLow?^` `.close^`).

Sample Applications

This section walks through **real programs** built in Final — not toy snippets, but small applications that combine language features you have already seen (SQL, tables, linear algebra, formatting) into a coherent story.

App	Status	Repo path
Auto-stats	Done for Stages 1–3 (wear → class → make)	<code>internal/final/apps/auto-stats/</code>
Interactive web tutorial	Planned	—

Auto-stats asks a practical question: *given a snapshot of used-car dealer listings, how well can we predict asking price from simple vehicle facts?* The narrative builds one regression ladder at a time — mean/age/miles, then vehicle class, then make — and reports honest baselines with PRE/F model comparison. The app sources live under `internal/final/apps/auto-stats/`.

The **interactive web tutorial** will be a separate sample app — a browser-based way to explore Final concepts step by step. Details are still to come; it is listed here so the book has a home for end-to-end examples as they land.

Start with [Auto-stats](#) if you want a data-and-stats story you can run today from the repo.

Auto-stats

Auto-stats predicts used-car **asking price** from listing data. In the repo the app lives under `internal/final/apps/auto-stats/`. Data comes from a frozen SQLite extract of dealer listings sourced from [unhuman.autos](#), stored as `internal/final/apps/auto-stats/db/auto-stats.db`.

This chapter walks three OLS stages on that freeze:

1. **Stage 1** — mean baseline, then age and miles (alone, then together)
2. **Stage 2** — vehicle class (car / SUV / truck) on top of age and miles
3. **Stage 3** — make (brand), still with age and miles, but **without** vehicle class

The question and the data

We want a number: *what should this listing cost?* A naive first answer is the **average price in the dataset** — anything fancier has to beat that on error (root-mean-square error, **RMSE**). We elaborate the model in stages, hoping each step predicts asking price better from facts such as age, mileage, class, and make. Alongside RMSE we report **R²** (correlation-style fit): **0** means no predictive value beyond the mean; **1** would be perfect prediction (rare in real markets). The goal at each step is **less error** and a **higher R²**.

Fit-ready rows for this chapter:

Filter	Value
Listings	n ≈ 2817 (2811 after class mapping)
As-of year	2026
Condition	used only (<code>inventory_type</code> null or <code>'used'</code>)
Miles	≥ 500 (drop near-new placeholders)
Outcome y	asking price (USD)

Age and miles **covary**: older cars tend to have more miles. Stage 1 therefore fits them **alone first**, then **together**, and compares nested models rather than jumping to a kitchen-sink equation. Later stages use the same compact-vs-augmented (**C-vs-A**) habit: keep a simpler model **C**, add a block in **A**, and ask how much of C's remaining error A removes.

PRE (*proportional reduction in error*) is that fraction: $(\mathrm{SSE}_C - \mathrm{SSE}_A) / \mathrm{SSE}_C$. An accompanying **F** statistic asks whether the PRE is large relative to the extra parameters and leftover noise.

Stage 1 — mean, age, and miles

Mean-only baseline

If every car were priced at the sample mean, $R^2 \approx 0$ by definition and **RMSE $\approx$ \$12,816**. The intercept is about **\$29,008** — the average asking price in this slice of the market. That wide scatter is the benchmark every richer model must improve on.

Age alone

One extra parameter (age in years) lifts **R^2 to 0.23** and cuts **RMSE to $\sim$ \$11,251**. The slope is about **−\$3,075 per year**: holding nothing else fixed, each additional year of age is associated with roughly three thousand dollars less on the sticker.

Compared to the mean-only model: **PRE $\approx$ 0.23, F $\approx$ 839** — age explains about 23% of what the mean left on the table.

Miles alone

Miles alone does better as a **single** predictor: **$R^2 = 0.33$, RMSE $\approx$ \$10,529**, slope about **−\$0.19 per mile**. vs mean: **PRE $\approx$ 0.33, F $\approx$ 1357**. **Better single = miles** — odometer reading alone beats age alone on this dataset.

Age and miles together

Adding both predictors reaches **$R^2 = 0.35$ and RMSE $\approx$ \$10,342** (three parameters: intercept, age, miles). Coefficients are roughly **\$43,289** intercept, **−\$1,279/year**, **−\$0.15/mile**. vs mean: **PRE $\approx$ 0.35, F $\approx$ 755** (two added df).

Incremental value after each single predictor:

Comparison	PRE	F
mean $\rightarrow$ age+miles	≈ 0.35	≈ 755
age $\rightarrow$ age+miles	≈ 0.16	≈ 517
miles $\rightarrow$ age+miles	≈ 0.036	≈ 104

Age still adds clear incremental PRE after miles — the two signals are related but not redundant. Even so, the joint model only nibbles another ~\$200 off RMSE compared to miles alone; most of the Stage-1 gain comes from getting odometer on the table.

What Stage 1 leaves on the floor

Results **do** beat the mean, but **residual RMSE is still ~\$10k**. Picture two listings with the same age and miles: a base Honda Civic and a lightly used luxury sedan can sit worlds apart in price. Age and miles explain broad depreciation; they do not know *what* the vehicle is.

That gap is the bridge to Stage 2.

Run: `internal/final/apps/auto-stats/ladder_age_miles.fnl` → word `ladder`.

Stage 2 — vehicle class (car / SUV / truck)

Age and miles still leave ~\$10k of RMSE. Intuition says **what kind of vehicle** matters: trucks tend to cost more than SUVs, which tend to cost more than passenger cars — even at similar age and miles. Stage 2 elaborates the Stage-1 wear core with a **vehicle-class** block.

Coding note

The listing feed's raw `vehicle_type` field is only **Car** or **Truck**, and **SUVs are lumped under Truck**. That collapses the story we care about. For this ladder we derive class from `body_type` instead:

Derived class	<code>body_type</code> values
car	Sedan, Coupe, Hatchback
suv	SUV, Wagon (e.g. Outback)
truck	Pickup

Unmapped body types are dropped (six rows here). Fit **n = 2811**. **Car** is the reference category; the design matrix adds **isSUV** and **isTruck** (with intercept). Compact **C** = age + miles; augmented **A** = age + miles + class.

Raw class means in this extract already line up with the narrative:

Class	n	Mean asking price
car	1078	≈ \$24,772
suv	1437	≈ \$30,517
truck	296	≈ \$37,370

Results — and how they match intuition

Model	p	R ²	RMSE
age+miles	3	≈ 0.35	≈ \$10,344
age+miles+class	5	≈ 0.43	≈ \$9,660

C-vs-A (wear → wear+class): **PRE ≈ 0.13**, **F ≈ 207** (df1 = 2). Class eats about **13%** of the error that age and miles left behind — a clear jump, not a rounding error. RMSE drops by roughly **\$700**.

Relative to a **car** of the same age and miles, the fitted level shifts are about **+\$4,600 for an SUV** and **+\$12,500 for a truck** (with age ≈ −\$1,285/yr and miles ≈ −\$0.15/mi still in the model).

That is exactly the ordering we expected going in:

1. **Raw means** already ranked truck » SUV » car (~\$37k / \$31k / \$25k).
2. **After controlling for age and miles**, the same ranking survives in the β s — so this is not just “trucks are newer” or “SUVs have fewer miles.” Class carries a **price-level** story of its own.
3. The **gap sizes** feel right too: SUV is a moderate step above car; full-size pickup is a much larger step — which matches how people shop those segments.

So the data-driven ladder does not overturn the garage-floor intuition; it **quantifies** it (PRE, dollars of shift, remaining RMSE) and confirms the shifts are still there once wear is in the model.

What Stage 2 leaves on the floor

Vehicle class helps — but **RMSE is still ~\$9.7k**. A nicely equipped **BMW 3 Series** can still ask more than a typical **RAV4** or **CR-V** at similar wear. Class is a coarse bucket; **brand and nameplate** sit inside those buckets. That motivates Stage 3.

Run: `internal/final/apps/auto-stats/ladder_vehicle_type.fnl` → word `ladderVt`.

Stage 3 — make (brand), without vehicle class

Luxury-vs-crossover prices show why class alone will not finish the job. Body class also **correlates strongly with make and model mix** (Ford's F-150 vs Honda's CR-V; BMW's 3 Series + X5 under one brand flag). So Stage 3 **drops vehicle class** from the design matrix and asks a different question: given age and miles, how much does **make** buy?

Honda is the reference make (a mass-market mid). Other catalog makes get one-hot columns. Compact **C** = age + miles; augmented **A** = age + miles + make. Fit **n = 2817**.

Results

Model	p	R ²	RMSE
age+miles	3	≈ 0.35	≈ \$10,342
age+miles+make	14	≈ 0.61	≈ \$8,013

C-vs-A (wear → wear+make): **PRE ≈ 0.40**, **F ≈ 171** (df1 = 11). Make removes about **40%** of the error left by age and miles — a much larger bite than class's ~13%. R² jumps from the mid-0.3s into the **0.60s**; RMSE falls from \$10.3k into the ****\$8k**** band.

Wear coefficients remain sensible with make in the model: about **−\$1,723 per year** and **−\$0.11 per mile**.

Make level shifts vs **Honda** (same age and miles), rounded:

Make	Δ asking \$	Make	Δ asking \$
RAM	≈ +\$21,800	Lincoln	≈ +\$17,500
Lexus	≈ +\$14,000	BMW	≈ +\$9,700
Ford	≈ +\$9,100	Dodge	≈ +\$7,200
Toyota	≈ +\$2,100	Audi	≈ +\$1,100
Subaru	≈ \$0	Chevrolet	≈ −\$400
Honda	reference	Nissan	≈ −\$5,600

How this meshes with intuition

- **Luxury and truck-heavy brands** sit well above Honda (Lincoln, Lexus, BMW, RAM, Ford-with-F-150 mix) — the BMW-vs-RAV4 story shows up as a brand-level premium even without a separate class column.

- **Near-Honda mass brands** (Toyota, Subaru, Chevrolet, Audi in this catalog mix) cluster near the reference.
- **Nissan** lands clearly cheaper at the same age and miles in this freeze — a discount the model is allowed to report without us forcing it.

One caveat, by design: a make dummy absorbs that brand's **model mix** (BMW = sedan + SUV; Ford = pickup + Explorer). That is why we removed class first — class and make are entangled, and Stage 3 asks for brand identity's predictive power on top of wear, not a pure "logo only" effect with class held fixed.

Where the ladder stands

Stage	Story	R ² (best in stage)	RMSE (best)
1	mean → age / miles	≈ 0.35	≈ \$10.3k
2	+ vehicle class	≈ 0.43	≈ \$9.7k
3	+ make (no class)	≈ 0.61	≈ \$8.0k

From "guess the average" (\$12.8k RMSE) to age+miles+make (\$8.0k), the ladder cuts a large share of error and more than doubles R² relative to wear alone — while still leaving a several-thousand-dollar cloud. Trim, options, condition, and exact nameplate are among the reasons; those are natural later elaborations if the catalog and *n* support them.

Run: `internal/final/apps/auto-stats/ladder_make.fnl` → word `ladderMake` .

Run it yourself

The three stage scripts live under `internal/final/apps/auto-stats/` .

From the **repository root**:

```
bin/final-go -core
"internal/final/apps/auto-stats/ladder_age_miles.fnl" compileFromFile
ladder

"internal/final/apps/auto-stats/ladder_vehicle_type.fnl" compileFromFile
ladderVt

"internal/final/apps/auto-stats/ladder_make.fnl" compileFromFile
ladderMake
```


Each script opens the frozen SQLite DB, builds design matrices, fits OLS models, and prints ladder lines plus C-vs-A PRE/F. Ingest and schema live alongside them if you want to trace data from JSON export to query.

Related language topics: [SQL Databases](#) for `DbProvider` and queries, [Table](#) for row-shaped results, and [Linear Algebra](#) (`VecNf` , `Mat2f` , `OLSModel`).

Contributors/Acknowledgements

The author would like to acknowledge the brilliance of Charles Moore, the inventor of Forth.

He would also like to acknowledge the brilliance of Henry Laxen and Michael Perry, the creators of Forth 83, an almost miraculous system that included a full 8086 assembler, and numerous other features, highlighting the power of Forth. Their “Laxen and Perry Forth” was an inspiration to the author.