How to Shiro

Complete Guide

Contents

[**Getting Started**](#_gjdgxs) **7**

[An Introduction to the Shiro Ecosystem](#_30j0zll) 7

[Why Learn Shiro?](#_1fob9te) 9

[What is this LISP Syntax I Keep Babbling About?](#_3znysh7) 10

[Let’s Play](#_2et92p0) 12

[Glossary](#_d8l4ef49i3ls) 14

[**Shiro From the Ground Up**](#_3dy6vkm) **18**

[Math, Comparison and Variables](#_1t3h5sf) 18

[Control Flow and Functions](#_4d34og8) 19

[Fun with Lists](#_2s8eyo1) 22

[Lambdas and Tigers and Bears](#_lrre6arz40t3) 23

[Objects, Implementers and Prototypes (or: The Art of Monkey-Patching)](#_3rdcrjn) 25

[Interacting with Nimue (the TCP/HTTP/Telnet server)](#_26in1rg) 28

[**Learn your Dev Environments: shiro and shIDE**](#_z7fmrjkswyj9) **32**

[shiro](#_6u41jl3p24px) 32

[shIDE](#_koxcvo6ilwtr) 32

[VS Code / Other](#_3aitpo4xyok8) 32

[**All the Keywords (ugh…)**](#_pda47xhhpwr1) **33**

[Normal](#_3c2ws7wcgip8) 33

[+ - / \* (Arithmetic Keywords)](#_17dp8vu) 33

[= != ! (Equality Testing Keywords)](#_7b617sjfmyw8) 33

[> < >= <= (Comparison Keywords)](#_i42t8r8s1sdl) 33

[.](#_lnxbz9) 34

[.?](#_35nkun2) 34

[.call / .c](#_g303ceci1uka) 34

[.s .set / .d .def / .sod](#_hwd7c08oivyn) 35

[apply](#_4phlihtn8li7) 35

[await / awaith](#_7p9wfq891qpw) 35

[concat](#_arthhjc7ny01) 36

[catch / throw](#_1aoup5254epx) 36

[contains](#_5ipbfn49rnkf) 37

[def](#_nw2l07lh463r) 37

[defn](#_z337ya) 37

[def?](#_3j2qqm3) 37

[dejson](#_1y810tw) 38

[do](#_4i7ojhp) 38

[enclose](#_8u6drx3rnsd) 38

[eval](#_dz470gdn2lc5) 39

[filter](#_1ci93xb) 39

[fn / =>](#_alzagro9d67f) 39

[fn?](#_4soupufflnxi) 40

[gv](#_2bn6wsx) 40

[http](#_3l18frh) 40

[if](#_9l1pcfp26rb8) 41

[implementer / impl](#_n7iyi5w3mbp6) 41

[impl?](#_pwn982fuig1l) 41

[import](#_rmvted4erq2a) 42

[interpolate](#_xyc43gndify9) 42

[json / jsonv](#_1pxezwc) 42

[kw](#_1ksv4uv) 43

[len](#_ftwgf4xkujf8) 43

[let](#_9ptvxzu3e4f1) 43

[list?](#_2p2csry) 44

[lower / upper](#_u7c04jo14sfd) 44

[map](#_y78icoof2x2o) 44

[mixin](#_x7efwuexvdek) 44

[nil?](#_pqnr99siw1t1) 45

[nop](#_23ckvvd) 45

[nth](#_ihv636) 45

[num?](#_32hioqz) 45

[obj?](#_1hmsyys) 46

[pair](#_41mghml) 46

[params](#_9bhzchzf1fk8) 47

[print](#_pdkz8qgelfr1) 47

[printnb / pnb](#_g96yl9c6kr4g) 47

[qnop](#_vx1227) 47

[quote / '](#_3fwokq0) 47

[range](#_1v1yuxt) 48

[relet](#_4f1mdlm) 48

[set](#_2u6wntf) 48

[skw](#_3tbugp1) 48

[sod](#_28h4qwu) 49

[split](#_ecqvpa7jza0p) 49

[str / string](#_wqpqhb9nwuj6) 49

[str?](#_37m2jsg) 49

[telnet / tcp](#_1kchj6od4bsa) 50

[try](#_391gm8c3tt1u) 50

[v](#_yruis8tw934v) 51

[while](#_dzi08kg4ofay) 51

[Contextual - Telnet / TCP](#_9v4lpzd0md5f) 52

[send](#_1rvwp1q) 52

[sendAll](#_4bvk7pj) 52

[sendTo](#_2r0uhxc) 52

[stop](#_er11rub8n28) 52

[Contextual - HTTP](#_csl232f6ka77) 53

[content](#_3ygebqi) 53

[rest](#_hpfu5ju0dfgn) 53

[route](#_filhidt6aifg) 53

[status](#_sqyw64) 54

[stop](#_7v192rlwaa64) 54

[Reader Shortcuts](#_aiwhrhfls4gr) 54

[Quoted Lists](#_ahpu20cq2m53) 54

[Dot Unwinding](#_d1uw4nh3u2q0) 55

[String Interpolation](#_oc97iusf1oji) 55

[AutoV](#_f3w6448ot2li) 55

[**The Shiro Standard Library (double ugh!)**](#_gt9gm91vdd83) **57**

[math](#_j43cgmpj4wgb) 57

[files](#_krgyl335266q) 57

[**Idiomatic Shiro**](#_lmeo1x1b9ooa) **58**

[With Great Power Comes Great Responsibility...](#_9ned0fxzbvoi) 58

[How Are You Going to Write Your Shiro?](#_3e94i2cb889f) 58

[Deciding How Much Safety You Want](#_72vivzvtw9qp) 59

[What Goes Where - Shiro Projects](#_hbhxim54t1r) 60

[Style and Structure](#_bb8s75ajm7x3) 60

[Bury your Branches](#_mrzwo1vcab04) 60

[Compose Lambdas for Patterns](#_tc1p799limw6) 61

[Code From the Inside Out](#_887zlg7a4dol) 61

[Use let-scopes and Namespace Objects](#_u7188c9217pd) 62

[Indentation and Blank Lines](#_ydvqircrzo10) 62

[When to Await](#_c9wlkdhm2nv) 64

[Implementers and Duck-Typing: Object-Oriented Shiro](#_6jb5i35z6o6i) 65

[Naming Conventions: My Suggested Approach](#_5n6ttgnpqazk) 66

[**Integrating with Shiro**](#_4pzh1qg4mcb4) **67**

[I Lied, Everything is Not a List, Everything is a Token](#_lxzzlm9igut8) 67

[Writing Shiro Libraries](#_bktg7051jvyh) 68

[Hosting the Interpreter](#_htmbijsvqj9f) 68

[**Putting it Together**](#_k7cofi7x36pt) **69**

[Writing a MUD in Shiro](#_kvfmie1lcakb) 69

[Writing a Persistent REST Service in Shiro](#_c0dnt7uyoq4z) 69

[Writing a Large, Batch-Processing Application in Shiro](#_g1jin96jrbq) 69

# Getting Started

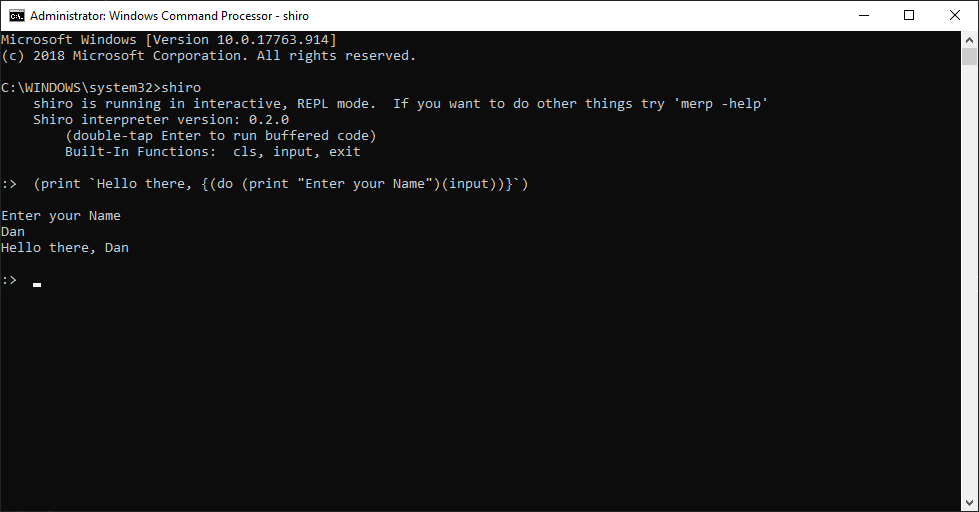
## An Introduction to the Shiro Ecosystem

Shiro is an interpreted, dynamic programming language (although it can be compiled, this is just really bundling the interpreter, runtime and your code together into an executable package) which looks a lot like LISP and behaves somewhat like it; I often refer to it as a LispScript. That sentence probably got rid of about 90% of the people who were considering learning Shiro, given that everyone hates dynamic languages and LISP-styled syntax. For the rest of you… welcome! Let’s begin.

The default Shiro distribution includes the following:

* shiro – a console application which can run Shiro, compile it, give you a REPL interface to play with it, install libraries and packages, etc.
* shIDE – A windows IDE which makes writing Shiro almost pleasant. It has most of the features you’d expect of a modern IDE, and adds a lot of special commands and options to help you write and read in the LISP style effectively.
* The Shiro Standard Library – A set of libraries which provide basic functionality (math, file system manipulation, web service calling, etc.) that are packaged with Shiro. The shiro console application can easily install any Standard Library package to your project.

If you’re an old console grognard, just go into your console and type ‘shiro’ and you can immediately start playing with the REPL. This is a good way to follow along with the code samples we’ll be using to explain certain concepts in a little bit – just be careful about copy-and-pasting. The shiro REPL executes when it encounters two Enters in a row, so if your sample has a double-line break in it you might accidentally execute if halfway through.

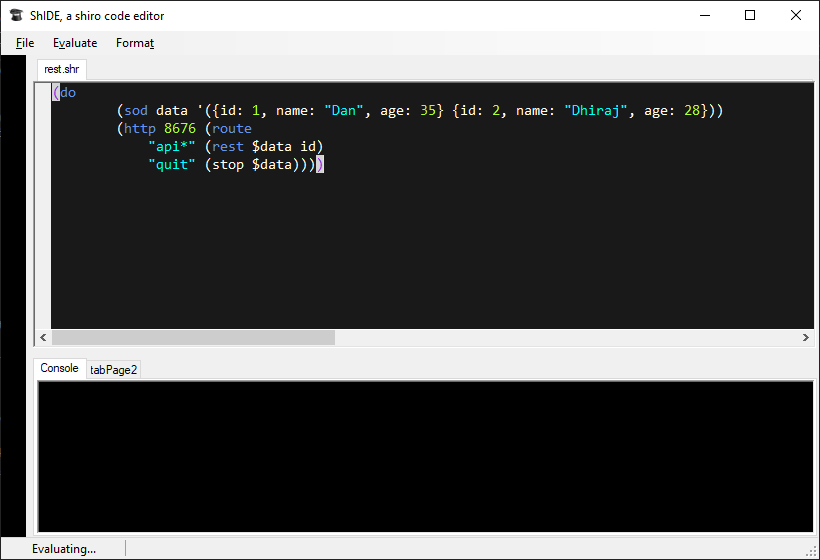


Running ‘shiro help’ will show you a list of commands you can use with it if you don’t just want to run the REPL. Some example commands:

* shiro compile main library -output:program.exe
* shiro run myfile -sr
* shiro install math
* shiro uninstall math

I’m sure I’ll eventually have to come back to more-fully document these, but for right now they’re in a bit of flux and ‘shiro help’ and some messing around should get you close enough.

If you prefer something that looks like it was made after 1998, shIDE is pretty much the way to go. The screenshot below was taken from a very early build of shIDE (you can tell by the lack of menus and that there’s no project tree over to the left), but should give you a general idea. Notice how you’ve got all kinds of nice stuff like syntax highlighting, brace matching (this is a godsend with LISP dialects), autocomplete and multi-editing.



Don’t worry, we’ll learn all about how awesome ShIDE is in a later chapter.

If you bothered to check out that code in the second picture, you might be a little intrigued. Surely that can’t be what it looks like, right? A fully functional REST server in five lines of code, without even importing any libraries or extra modules? That’s like, unheard of in a programming language – seriously, try it in node. You’re 5 libraries, multiple source files and a hundred lines of code deep before you even get started.

Which brings up a good point…

## Why Learn Shiro?

Shiro was designed very specifically to do certain things very quickly and efficiently – and I mean that both in terms of how long it takes you to write the code, and how the result performs. At the heart of it is a lightning-fast, hand-coded TCP/IP server called Nimue which can speak rudimentary HTTP, telnet, or raw TCP/IP; it is applications which can best use this piece that you might want to consider writing in Shiro. So, if you want to stand up a small REST microservice, or a TCP/IP websocket server, or a Telnet command parser, you can’t go wrong doing it in Shiro. The result doesn’t need IIS, complicated third party libraries, server deployment or any of the .NET HTTP runtimes.

Shiro thrives in DevOps as well, creating small utilities, scripts and services which can automate annoying processes. In addition, the rapid time-to-development means that some DevOps projects that involve web services and complex integrations can be done in hours in Shiro instead of whole sprints.

Shiro is also pretty fun to write. In a way it’s like Scala (cue angry Scala nerds storming my condo) because it offers multiple programming paradigms simultaneously. You’ve got your expression-tree based LISP syntax, but you’ve also got JavaScript style objects which have some pretty interesting and advanced OO concepts available to them. Its highly functional (of course it is, LISP invented functional programming), but the nature and structure of it makes it less intimidating than many functional languages, and its dynamic, permissive syntax lets you do some really neat things.

I’ve personally used Shiro in real-world, work-related applications in a number of ways:

* Mocking backend and BFFE services so that I can do front-end web development without waiting on the back-end guys to get their shit together.
* Writing devops validation services which were able to quickly validate deployments of our software in over a thousand locations and plug right in to a Jenkins pipeline.
* Lightweight microservices with limited integration to the larger ecosystem.
* A rather interesting TCP/IP proxy for a mobile application.

And that’s dealing with the barrier that always exists when you say, “Hey guys, I wrote this LISP dialect we could *totally* do this in really fast!”

In addition (and I promise I won’t mention this again for the whole rest of the guide), learning LISP syntax makes you a better programmer. LISP (and thus, Shiro) is basically just a written-out expression tree, which is what your compiler/interpreter of choice is turning your code into anyway. By stripping away literally all the syntax you get right at the heart of what coding actually *is*. Learning LISP back in the day made me a better C++ programmer, and nowadays even though I write C# and JavaScript for a living I still owe a lot of my understanding of high-level concept to LISP.

Okay, evangelism over. On to the learning…

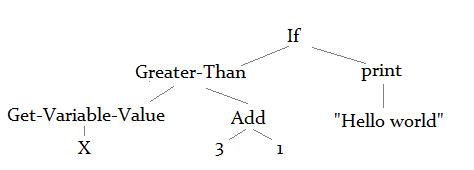
## What is this LISP Syntax I Keep Babbling About?

Under the cover of your favorite compiler or interpreter of choice is probably something called an expression tree. The early steps of processing a programming language involve turning your syntax into a tree that can be easily walked and executed or turned into bytecode of some kind. Let’s look at a simple programming construct in a hypothetical language:

if(x > 3+1)

print 'Hello World'

After intermediate processing by the interpreter or compiler, this would be turned into a tree-like structure in memory, which might look something like this:



If we were to “walk” this tree in our minds, we’d encounter the ‘if’ command, then go down the first branch and end up comparing x to 3+1 just as in our original code example, then if the result were true we print hello world. The idea is that the deepest leaf of the tree should be a constant value of some kind (a name, a value, etc.), while higher nodes in the tree represent commands and transformations.

Now it takes quite a bit of work to turn something like C# or JavaScript into a tree like this – believe me I’ve written my share of compilers and interpreters, using both hand-coded parsers and generation tools like Bison and Yacc. It would seem smart then, especially if you’re a lazy person like me, to make the syntax of your programming language match this tree structure as closely as possible.

Enter LISP. Let’s write out that code as a text-based representation of the expression tree (note: this is not quite syntactically-correct Shiro, but only because I put ‘value-of’ instead of ‘v’ so you would know what it meant):

(if  
 (>  
 (value-of x)  
 (+ 3 1))  
 (print 'Hello World'))

Hmm, this is starting to look a little bit familiar, you might start to see where I’m going with this.

If you accept my parenthesis-based way of textually representing an expression tree, you can see that certain structure is imposed on the resulting code by the nature of the tree. Every sequence (or ‘list’ as you might call it) begins with a command, then takes 1 or more parameters, which themselves can be lists. Any given list is evaluated by evaluating the inner parameter lists first, then passing the resulting values to whatever command or keyword is at the beginning of the list.

Congratulations, you can now at least read LISP and figure out what you’re looking at!

## Let’s Play

Bring up either the Shiro REPL or ShIDE for this part… we’re going to start playing around, typing code and figuring out how this thing actually works. We will begin where every programming language tutorial in history begins, except with a nod to Animaniacs. Type this into your editor of choice and run it:

print 'Hello Nurse!'

If you've ever programmed anything before you probably had a good idea what was going to happen, and lo-and-behold, it happened! But what was all that crap I said about everything being a list? That's not a list, it's just “print hello world”, the same thing you write in every other language whose print isn't a function call! Have I already been lying to you? Nope. You see, Shiro likes to be helpful, so it will wrap your top-level commands in a list for you if you forget. The actual, syntactically-correct way to do the above is this:

(print 'Hello Nurse!')

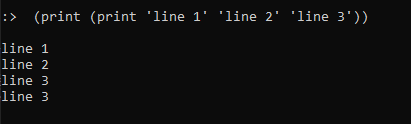
Which if you type and evaluate you'll see has the same result. The parentheses tell Shiro that we're starting a list, and then everything in it is parsed into a separate element. It’s just a simple, text-based representation of that tree-structure we talked about in the previous section.

The first thing in a list in Shiro is called either a command. Commands can be a lot of different things, but the most common thing is a keyword, or built-in function of the language. That’s what we have here… print is a keyword. Lists are evaluated in Shiro by looking at the command, passing it the parameters (the rest of the stuff in the list), and then doing whatever that command is supposed to do. print is a fun keyword because it can take any number of arguments, so you can print a bunch of lines pretty easily:

(print 'This is a line' 'And this is a line' 'Guess what this is?')

We’re rolling right along now; evaluating a single list (that is, a list that doesn’t have other lists in it) is nice and easy to wrap your mind around. By the way, have you wondered why I always call it ‘evaluating’ a list, a not ‘executing’ a list? It’s an important distinction to make, because every list in Shiro will end up producing some kind of value (ie: “evaluating” to something)). Try typing this:

(print (print 'line 1' 'line 2' 'line 3'))

If you’re following along, you might’ve just figured out that the print keyword always evaluates to the value of the last thing it printed. As you can see in the picture over to the right, we evaluate the innermost list, print our three lines, then evaluate the outermost list, which prints the result of the innermost list, which is the third line.

In the event that there’s just *nothing* sensible to return from evaluating a list, we return a special value called Nil, which is basically a fancy way of saying “nothing”. But in most cases, you’ll find that lists evaluate to a particular value, and to a value that’s been chosen to make it easy for you to construct your lists. You won’t often use the result of evaluating a print command, but it’s a simple way to get the concept in your head.

Let’s say I want to print a list. I’d probably try it like this:

(print (1 2 3)) ; wrong (also this is how you make comments)

And as the comment implies, I'd be very, very wrong to do so and Shiro will yell at me about it. Can you see why? You have all the information you need to figure it out… For an extra hint try running it and see what the error message says.

Earlier I said that first thing in a list in Shiro is the command, the thing that tells the list how to evaluate. 1, which is the first thing in the list we’re trying to print, is not a thing Shiro knows how to evaluate – which is to say it’s not a command or a function or anything like that. So when Shiro got to that innermost list and went to evaluate it to figure out what it was printing, it couldn’t.

But Dan, you say, programming is all about dealing with lists of data! What is the point of list-based programming language without usable data lists? Well, say hello to the quote keyword,

(print (quote 1 2 3))

If you try to evaluate that, it works and has the results you were probably expecting. It also perhaps fills you with a sense of foreboding at how many times you’re going to be type “quote”, and how that’s going to munge up your code. Fortunately, Shiro provides a reader shortcut (basically a shorthand way of typing something) for quoted lists. You can also write the above code like this:

(print '(1 2 3))

Which is much better to type. And you’re welcome by the way because that presents some interesting lexical ambiguities that cost me literally hours while trying to get syntax highlighting working in ShIDE.

Every Shiro file (including something you try to evaluate on the REPL) is a single list, always. If you tried to do something like the below (which is wrong on so many levels it hurt me to type the sample), you’d have problems:

(print 'Hello world')

(print 'Oh yeah and also hello universe')

If you try it, you get the most instructive error message of all time (well, not really. Shiro is big on instructive, helpful error messages):

[error] Sibling peered list passed for evaluation – you are probably missing a 'do' keyword

As the error implies, you need to turn your multiple-lists into a single list, and you can use the do keyword for that. I'm sure you already figured it out, but it looks like this:

(do

(print 'hello world')

(print 'oh yeah and hello universe too'))

Like print, do evaluates to whatever the last thing in it evaluates to.

Strings are pretty cool in Shiro. You can use either double or single quotes for them (allowing you to use the other kind of quote inside the string). They can also have line breaks in them. There are several escape characters you can use in a string, like %n (newline). We'll list them all out later. Here's a little snippet that shows off a few of these attributes:

(print "Strings can have%s

line breaks in them , and also%s

can include the 'other kind' of quote. You can escape the%s

quote you used for the string %"like this%"")

Like every cool programming language on the block, there’s also built-in string interpolation in Shiro, which has both an ugly way to do it, and a nice, easy reader shortcut. Here’s an example written both ways:

(do

(print (interpolate "2 + 2 = {(+ 2 2)}")) ; ugly

(print `2 + 2 = {(+ 2 2)}`)) ; reader shortcut

The weird tick-mark we use for the reader-shortcut version is the one on the tilde (~) key of your keyboard. I have no idea what it’s even actually for, so it seemed like a safe bet.

Alright, that's literally all you need to know to start learning Shiro. There aren't any other weird syntaxes you need to learn, no other rules of grammar. You just make lists, and then they evaluate each other and ultimately result in doing something.

## Glossary

While most of the terms used in this guide are in a general programming context, some of them are Shiro-specific and may not exactly map with how those terms are used in other contexts (usually in a LISP context). For example, our definition of a ‘reader’ would probably give a LISP purist some kind of seizure. Still, these terms will be used throughout this guide and you should be familiar with them.

**Async-List:** A type of list which is evaluated asynchronously (using the await keyword). Has its own local copy of the symbol and function tables which is destroyed when it finishes evaluating without changing the main interpreter’s symbols and functions. Any shiro command can be awaited.

**Auto-Predicate:** Every time you create an *implementer* you also get a free *predicate* to check if something implements that implementer (saving you from having to use the impl? keyword). For example, if you make an *implementer* called IDoStuff, you also get a free *predicate* called IDoStuff?.

**Command:** A command is the first item in a Shiro *list*. It can be either a *keyword* or a function name.

**Contextual Keyword:** A type of *keyword* which can only be evaluated in a specific context. Usually this is a keyword that is specific to a *Nimue* server mode. Attempting to evaluate a contextual keyword outside the appropriate context will result in an error.

**Enclosure:** An *enclosure* is a special part of an object for stodgy, boring people where they can hide variables and methods they do not want to expose outside the object. Basically equivalent to private/protected elements in other languages. This is one of those ‘you can do it if you really want to’ kind of things.

**Hermetic Await:** A special kind of await which is more performant than the normal kind done with the await *keyword*. *Hermetic awaits* do not copy the local interpreters symbol table (so variables and functions and implementers are not copied) and the awaited evaluation occurs in a completely fresh interpreter. This cuts back significantly on the performance overhead to starting up a new interpreter for awaiting, especially in applications with large and complex symbol tables.

**Implementer:** A special type of object in shiro which represents a piece of functionality which can be brought wholesale into other objects or implemented within that object. They’re kind of like interfaces in static, classical-inheritance languages, but they also sometimes come with functionality. They can be used a classes, interfaces and plugins, sometimes all three at once. Sometimes also called *Mix-Ins* or *mixins*.

**Implicit Quote:** A *list* in Shiro always starts with a *command*, telling it what to do. There is a keyword called *quote* and a *reader shortcut* for it that will create a list that is not evaluated and instead simply contains data. Should Shiro try to evaluate this list it will create an error (unless the first item in the list happens to be a valid *command)*, but if the code in question is intended to handle data lists this can allow you to pass sequences around as you might an array or linked-list in another programming language. Generally you explicitly quote your lists like this:

(quote 1 2 3 4)

-or-

'(1 2 3 4)

But in some cases Shiro knows that it’s expecting a quoted list (for example, in the list of arguments in a function definition, or the name:value pairs in a let-scope), and in those cases it will create an *implicit quote* for you, effectively treating the list as quoted even if you do not quote it. You should be careful not to quote implicit-quoted lists, or you will end up with an extra item in them.

**Inline Object:** An inline-object is a bit of JSON that is treated as a special kind of *list* in Shiro. Inline objects are *lists* with an *implicit quote* wherein all the elements of the list have names attached to them. For example:

{name: "Dan", age: 35}

Is effectively a Shiro *list* with a quote (so, a list that will not be evaluated) with two items in it, “Dan” and 35. Each of these items has a name attached to it (name and age respectively).

**Interop Function / Interop Variable:** An interop function or variable is a type of function (or variable, duh) which is injected into Shiro from whatever is hosting it. Functions like cls and input, which are written in C# and injected into the Shiro runtime are examples. **Any** implementation of Shiro can inject their own interops, and no consistent set of interop functions is expected or provided. Shiro libraries are also just a series of interop functions and variables. Once installed they perform just as well as native language keywords. Sometimes also called *autofunctions* and *autovars*.

**Keyword:** A keyword is a type of *command* which is built in to Shiro, representing the core building blocks of your code. Most keywords (like *print*, *if*, or *sod*) can be used anywhere. Some few keywords (usually those related to network-server functionality like route, status and sendAll) are called *contextual keywords* and can only be used in certain contexts.

**Let-Scope:** Most of the variables in Shiro are global variables. It is, however, possible to create a local-variable scope. In most languages local scopes are created for you at various scope points (like inside a function), but in Shiro you usually create them yourself using the *let* keyword. The list within *let* will be evaluated with the local variables in place, and they will then be cleaned up once that list is done evaluating. These local scopes are called *let-scopes*.

Some situations and keywords in Shiro create an implicit let-scope (for example, within a network server’s handler list, there are usually let-scoped variables like id, request and input which you can access, and which will automatically clean themselves up).

**List**: A list in Shiro is a sequence of elements, with the first element being a *command* and the rest of the elements (if any) being *parameters*. Any parameter can itself be a list, which will be evaluated to determine its value. Our favorite code example,

(print 'Hello Nurse!')

is a list, with the command ‘print’ and the single parameter “Hello Nurse”. While it is fairly easy to map lists to arrays or linked-lists in other programming languages, they work a little differently in a LISP-like language, because they can be both code and data at the same time. More on this later.

**Mix-In / Mixin:** See *Implementer*.

**Nimue:** Nimue is Shiro’s TCP/IP server infrastructure. It is built directly into the interpreter and extremely low level, designed to be incredibly fast and lightweight. It has modes which can handle Telnet and basic HTTP built in, and can have any TCP/IP-based protocol implemented on it easily enough. Though Nimue is highly threaded it executes all of your Shiro in a thread-safe way.

**Parameter:** All of the items which follow the *command* in a Shiro *list* are the parameters of that *list*.

**Parameter Predicate / Predicate Parameter:** Parameters to functions and lambdas can have predicates attached to them which will evaluated to determine if the call is valid or not. These are called param-predicates or predicate parameters.

**Predicate:** A predicate in Shiro is a specific type of command which returns a true or false is used for conditional checks and filtering. As a general rule, predicates end with a ?. Some examples of Shiro predicates are: str? num? def? nil?. You can implement your own as *interop functions*.

**Reader Shortcut:** The first thing done when you evaluate Shiro code is that it goes through a reader, which transforms it from a string into a data structure which more closely mirrors the Shiro *list*. This reader has a few helpful shortcuts, wherein if it encounters a certain syntax it will automatically translate it into a certain kind of *list* for you. For example, the $ syntax used to access variables:

$x

-is turned by the reader into-

(v x)

All of these odd syntactical helpers are referred to as *reader shortcuts*.

# Shiro From the Ground Up

I’m assuming you’re already somewhat familiar with programming, so that we can move fast and not have to build up each concept from the ground up. If (somehow) Shiro is your first programming language then you’re going to have to do a lot of playing around, reading between the lines and re-reading this section to follow along. There is a detailed keyword reference later on to give you more complete information, but it’s presented alphabetically so you really have to at least understand the categories of thing we’re talking about in this section to make good use of the list.

As ever, keep a REPL or a ShIDE open and mess around with our sample code as we go. This is by far the best way to learn Shiro quickly.

## Math, Comparison and Variables

Variables are pretty standard in shiro. Note that I've included a few extra line breaks for clarity in this code sample, which is fine if you're using the interpreter or compiler, but will cause the REPL to try and evaluate half a list and complain.

(do

(def x 1) ; declare x for the first time and set it to 1.

; A variable can only be defined once

(set x 5) ; set existing variable x to value 5

(sod y 10) ; 'sod' makes it easy to work with variables, by

; using either set or define (get it? s.o.d)

(sod y 23) ; see?

; You need to explicitly get the value of a variable:

(print (str "x = " (v x)))

; ... but there is a reader shortcut to do this using $:

(print (str "y = " $y)))

So all those variables are global. There is only ever one 'x' per shiro instance. If you're using the REPL, it will persist throughout the entire session, otherwise it will exist in all your different code files because they share the same instance of the shiro runtime. You can, however, create your own local variables whenever you want by making a scope level. You do that with the let keyword:

(do

(def i 255)

(let

(i 1 j 2)

(do

(print $i) (print $j)))

(print (def? i) (def? j) $i))

So what are we doing here? First, making a global variable called i and setting it to 255. We'll use this fact later, so keep it in mind. Then we encounter our let keyword. Let takes two arguments, and the first one is somewhat special because it has what's called an implicit quote. As you note it's a quoted list ('i' is not a shiro command), but we didn't have to quote it. I figured I'd save you the keypress. This first list must have an even number of things in it. They are basically paired, with the first being the variable name and the second the default value.

Variables in a let-scope hide global variables, so i inside the let is 1, not 255. Once we leave the let-scope the global i is unhidden and retains its original value. You'll notice (at least, if you will if you figured out that that def? keyword returns true if something is defined and false otherwise) that variables inside a let-scope are destroyed when the scope ends, so j ceases to exist.

You can do all the ordinary kinds of math and comparison that you're used to in other programming languages, but you do it using the rules of shiro syntax, so the command (or in this case, the operator) goes at the beginning of the list. Here are some examples:

(+ 2 2) ; 4

(+ 2 (- 3 1)) ; 4

(+ 3 3 3) ; 9

(= 2 2) ; True

(= 2 2.5) ; False

(= 2 (/ 4 2)) ; True

(= nil "nil") ; False

(! true) ; False

(! nil) ; True

(! 0) ; True

(> 3 2) ; True

(>= 2 (+ 1 1)) ; True

Most of this is pretty straightforward, about the only really interesting things to note is nil, which is a particular value in shiro that means “nothing”, it's like NULL in other languages. Also note that shiro has truthiness like JavaScript, so you can use numbers, objects or even strings as booleans without incident – at least without incident if you knew you were doing it and intended to.

## Control Flow and Functions

Now that you know shiro has booleans (duh), you can probably also guess it has ways to branch based on them. To do so we use the innovative keyword if:

(if true (print "Hello world"))

(if false (print "Won't Print") (print "Will Print"))

Remember how I said that every list in shiro evaluates to something? Well because of that property, the if keyword can also be used just like a ternary operator (the ? : in most languages). Like so:

(print (if false "Won't Print" "Will Print"))

And of course those strings could be lists as well, and if you keep extrapolating that you're programming in shiro! You can loop in shiro (while loops at least), although you're being kind of weird most of the time if you do so because there are much better ways to do it like the map, filter and apply keywords we'll learn about later. But if you want to be weird, here's a while loop in shiro:

(do

(sod x 10)

(while (> $x 0) (do

(print $x)

(set x (- $x 1)))))

Stunning, right?

So far we’re using pretty simple conditions (like mathematical comparisons and whatnot), but there are lots of more interesting things you can do. Shiro includes a ton of predicates -- keywords which evaluate to boolean values and are intended for use with control flow elements. Predicates end with ‘?’ by convention (although you can make your own that don’t if you want, I won’t stop you).

(sod x '(1 2 3))

(sod y 2)

(sod z {name: "dan", age: 36}) ; More on this later, don't panic!

(sod s "Hello nurse")

(list? $x) (list? $z) ; True

(list? $y) (list? $s) ; False

(obj? $z) ; True

(obj? $x) (obj? $y) (obj? $s) ; False

(num? $y) ; True

(num? $x) (num? $z) (num? $s) ; False

(def? x) (def? s) ; True

(def? bob) ; False

(fn? (=> (print 123))) ; True

(fn? $z) ; False

Don’t let the preceding section fool you into thinking shiro is a strongly-typed language or anything like that, but we can at least tell what the thing we’re looking at is at runtime.

Speaking of things that are likely to cause errors… shiro has try, catch and throw keywords, but they work a little bit differently than you might expect from other programming languages. try and catch are sort of sibling keywords -- try is a superset of catch. catch will only catch thrown exceptions (exceptions you specifically throw with the throw keyword), while try handles what would normally be a parser error and thrown exceptions. For example this first block runs without incident despite a “sibling peered list” error in the code, because it uses try. The second block will fail, because it uses catch:

;block 1 - this works. try eats the list-pairing exception

(try ((+ 2 2)(+ 2 2)) "this will be the result")

;block 2 - this will cause an error because we didn’t throw anything

(catch ((+ 2 2)(+ 2 2)) "this will not be the result")

;block 3 - catch with throw (and a final-list)

(catch (throw 123)

(print (if (= $ex 123) 'this will print'))

(print 'this also happens'))

There’s a lot of information hiding in these three snippets. In the third one you can see that within the list evaluating if there is an error (this applies to both try and catch) there is an ex variable which contains whatever was thrown. If you ran the third example, you probably also noticed that the result of the overall list is the final-list if there is one -- otherwise it would be the error-hander list if there was an error or the result of the first list if not. The basic format for try and catch then, is:

try/catch

(list to try and evaluate)

(list to evaluate if there is an error/throw)

(optional list that will be evaluated in either case)

Note that in the finally-list there is a variable called result which contains either the result of the first or second list -- this allows you to “pass through” the value after doing whatever cleanup you might want to do in the final-list.

Shiro has functions (boy oh boy does shiro have functions!). The least interesting kind are just... well functions. You define them, they have names, and you call them just like everything else in shiro, by putting the name of your function as the list’s command.

(defn say-hi (name)

(print `Hello {$name}`))

Pretty basic stuff. Define a function named say-hi, which takes one parameter called 'name', then says hello to the name. You call it like any other first-class shiro command:

(say-hi Dan)

For those of you that feel comfortable with a little bit of type safety in your lives, Shiro provides Parameter Predicates, which is a special kind of function parameter which has a predicate (one of those commands with a ? at the end of it) attached to it. The function call will only work if the parameters passed in match the predicate. So if we only wanted to be able to say hello to people whose names were strings, we could do this:

(defn say-hi (name:str?)

(print `Hello {$name}`))

And now you can say-hi “dan”, but not say-hi 123.

And that’s pretty much all there is to functions (... he said, rubbing his hands together with glee and cackling), at least for now. But seriously, that’s not all there is to functions and we’ll be breaking our brains together in the later section on lambdas.

## Fun with Lists

If you're still reading and understanding, you're probably starting to get shiro a little bit even if you don't have a background with this sort of syntax. Everything's a list, often a list of lists, and we just sort of evaluate them from the innermost lists to the outermost ones until we get a final result. Cool.

Since shiro is a programming language where everything is a list, there are a bajillion ways you can manipulate lists using different commands and functions. And since every list is technically also code, you can use these functions to dynamically build executable shiro and it's no different from the code you'd write to manipulate a list.

When you're making a list, remember the basic rule – the first thing in the list is the command unless the list is quoted (in which case this is still true, but the interpreter sneaks a 'quote' keyword in there for you). So when you're making a list, if you want a list that's purely data (like an array or linked-list type thing) then you want to make sure it's quoted, otherwise you might be accidentally building code that shiro will try to evaluate. This almost always manifests at runtime as an “Unknown Keyword” error.

There are lots of ways to slice and dice lists to your needs (and I do mean 'lots'). You can get the keyword of a list (the first thing in it) with the 'kw' keyword, and you can get the rest of the list with the params keyword. Here are some examples of those two and others simple ways to get stuff out of lists:

(do

(print (kw '(1 2 3))) ; 1

(print (params '(1 2 3))) ; 2 3

(print (nth 2 '(1 2 3))) ; 2

(print (range 2 2 '(1 2 3 4)))) ; 2 3

But really you very rarely want to slice lists up this way, and when you do you're either doing something very boring, or very interesting like making dynamic code at runtime. It's a lot more interesting to do things to stuff in lists. A lot of the time you use a for loop or a foreach loop in your programming language of choice to iterate through a list; you do that same stuff in shiro, but of course it's different. We’ll get into these kinds of keywords more in our lambda section below, but for now here’s a quick example,

(do

(sod stuff '(1 12.5 'Dan' ))

(print (filter num? $stuff))) ; (1 12.5)

filter in this case is a keyword that says “evaluate and return every item in the input list that matches the predicate in the first parameter”. You could do the same thing with a while loop if you wanted, but it would be much slower, uglier and unidiomatic.

You can put lists together with the concat keyword:

(concat '(1 2) '(3 4)) ; '(1 2 3 4)

This can result in an evaluable-list (ie: a non-quoted list) in some cases, and in some cases you might want to explicitly evaluate them, which you can do with the eval keyword. You use eval pretty rarely (because shiro automatically evaluates lists in 99% of cases), but if you’re building dynamic code it can come in handy. Here are some ways to use eval:

(eval '(print 'hello world')) ; note that the inner list is quoted

(do

(sod l '(1 2 3))

(eval (skw print $l))

(eval (concat print $l)))

Many (boring) people write code their whole lives without ever generating code at runtime to execute, but if you want to be one of the interesting ones, shiro makes it as easy as it can be.

## Lambdas and Tigers and Bears

Functions are okay I guess... they basically let you make your own language keywords, which is neat, but they're so static and monolithic and boring, it would be much cooler if there were functions that weren't named anything and were just passed around like values...

Good News Everyone! There is a type of function just like that, called a lambda or anonymous function. We can make one that works a lot like say-hi above by doing this:

(sod say-hi2 (fn s (print `Hello {$s}`)))

Paste that hideous, chthonic gibberish into the REPL, then try typing 'say-hi2 Dan' again and lo and behold, it works just the same. The reason for that is that we created a variable named say-hi2 in that snippet and actually assigned a function to it. The 'fn' keyword creates a lambda, with the first parameter being the argument list and the second the body (you can omit the argument list if the lambda doesn’t take any parameters). You can also use => instead of fn as the keyword if you like making sure people can't read your code.

Now if all you could do with lambdas is assign them to variables and call them just like functions they'd just be functions with extra steps and slightly less efficiency at runtime. Fortunately, there's so much more you can do, like passing them as parameters to other functions, or keywords. For example there's a keyword apply in shiro which applies a particular command to everything in a list. You can use it like this:

(apply print '(1 2 3))

Don't worry we'll be talking that stuff to death a bit later on. For now it's good to understand that you can also use a lambda as the first parameters of apply (or any similar command):

(apply

(fn s (print $s))

'(1 2 3))

Lambdas by themselves are even perfectly valid as commands, however they come to be in the first position of a list. Here's a very ugly and obscure way to calculate 2+2 in shiro:

((=> (x y) (+ $x $y)) 2 2) ; note => and fn are interchangeable

If you're not confused by that then I must be doing a really good job describing Lisp syntax. Basically the first item of this list is a list that evaluates to a lambda, which is something shiro knows how to treat as a command. The next 2 parameters (2 and 2) are the parameters to the lambda.

This might seem like something you would never do explicitly like this (and generally it’s not, although sometimes you might evaluate a list that evaluates a lambda that you then want to evaluate… shiro gets weird like that), but you’d be surprised how powerful it can be when you get rolling with it.

Once you start getting comfortable passing lambdas around as parameters to other commands you start to unlock the Functional Programming Novice Achievement. Shiro has a bunch of keywords designed for you to do just that. Remember when we talked about the while loop and I said loops are lame in shiro? This kind of stuff is why. Rather than looping through a list you’re much better off filtering it, or applying a lambda to each item in it. Some examples for you:

(filter (=> (n) (> $n 5)) '(1 10 7 3 -4 154)) ; '(10 7 154)

(filter num? '(1 2 'Dan was here' 123.5)) ; '(1 2 123.5)

; Notice the difference between:

(map (=> (x) (+ $x 1)) '(1 2 3 4)) ; '(1 2 3 4)

(apply (=> (x) (+ $x 1)) '(1 2 3 4)) ; '(2 3 4 5)

As you can see from the second filter example, filter can also take a predicate (or indeed anything that results in a boolean and takes in a single parameter). map and apply are two similar keywords, with the only difference being that apply actually applies the result of the lambda to the resulting list, while map just evaluates the lambda and leaves the original list intact. map and apply can also take any valid command as their first parameter -- it doesn’t *have* to be a lambda, it just can be.

Just like with functions, you can use predicate-parameters to add a measure of type-safety to your lambdas.

## Objects, Implementers and Prototypes (or: The Art of Monkey-Patching)

You may have noticed a few times that we sometimes make what look like JavaScript objects in shiro. They do a lot of what you'd expect a JavaScript object to do. You make them like this:

(sod o {name: 'Dan', age: 36, loc: 'OR' })

And then you can:

(do

(print (. $o name))

(.sod o name 'Steve')

(print (. $o name)))

Notice that the dots work a lot like dots in normal languages, they just use shiro syntax instead of the more traditional one. You can dereference down any number of layers with a single dot, so if you have objects containing objects containing objects you can get even to the innermost properties with a single list. Keywords like .sod in the example above (and the obvious counterpoints .def and .set) can be used to change and create new properties on objects. If you're not sure if a particular object has a particular property, you can use the .? command, which returns nil if it can't find any of the properties you ask for.

Now I can already feel you wincing at that . keyword, so we have a reader shortcut in place for it. The above snippet can also be written:

(do

(print o.name)

(.sod o name 'Steve')

(print o.name))

You’ll still need to use .sod and those kinds of keywords to *set* values, but you can get values using the normal dot-notation that every other programming language uses and shiro’s reader will unwind it for you.

Now objects in shiro are just lists (everything in shiro is a list, he said for the 100th time), but they have a special property wherein the values in the list have names. These are called pairs. You can make a pair using the pair keyword, which you can use as a backhanded way of adding things to objects if you don't want to use .sod for some reason. Check it out:

(do

(sod obj {name: 'dan'})

(print (.? $obj fakeProperty)) ; nil

(sod obj (concat $obj (pair fakeProperty "Its magic!")))

(print (.? $obj fakeProperty))) ; "It's magic!"

So we make an object with a single property (name), prove that there's no property named fakeProperty on that object, then we add a new pair to obj using concat and pair to make a new named value. Then we prove that the new value is there. This bit is just here to help you understand a bit about objects, pair is one of those keywords like eval that’s used pretty rarely but has niche applications. The correct way to do what you’re doing in the example above is using .sod.

Since we already know about lambdas, I bet you figured out that objects can have lambdas as properties.

(do

(sod o {

say-hi: (=> (print 'hello nurse')),

say-hi-to: (=> s (print `hello {$s}`))

})

(o.say-hi)

(o.say-hi-to "Dan"))

Now we’re cooking with gas! We’ve got lists, lambdas, objects, tuples… you name it. Beneath the surface it’s all shiro lists and it all fits within this strange, LispScript paradigm we’re making up here, but you’ve got access to all the fun programming constructs, and even some of the stodgy ones.

If you’re expecting this is where I’m going to start talking about classes, then I’m about to disappoint you. Shiro doesn’t do classical inheritance, so there are no classes. We have something called implementers instead, which are some combination of class, interface and plugin depending on how you use them.

Implementers are basically objects, but they’re special objects in that once they’re created and defined they can’t be munged and manipulated (ie: no .sod or concatting things onto them or whatnot). In that sense they’re *almost* like classes. When an object implements an implementer, it brings that implementer into itself automatically unless it’s already done something to override that implementer. In that sense, they’re *almost* like interfaces.

Confused? That’s probably for the best. An example should help, at least a little. Let’s imagine a simple implementer called IPrintMyself which… well, prints itself. Here’s how it looks:

(do

(implementer IPrintMyself { print-myself: (=> (print $this))})

(sod o (mixin IPrintMyself { name: "Dan", age: 36}))

(o.print-myself))

The implementer keyword (which can be abbreviated as ‘impl’ if you’d like) defines a new implementer. Implementers can have all the stuff objects can -- properties and methods basically. You can’t access them directly though… if you tried to print $IPrintMyself it wouldn’t work, because IPrintMyself isn’t a symbol or an object.

What you can do with implementers (as seen in the next line where we use the mixin keyword) is mix them into other objects. Any number of implementers can be mixed in (and you can mix things in to objects that have already had other things mixed into them later on of course). You can check and see if an object implements a particular implementer using the impl? predicate:

(do

(sod o {name: "Dan", age: 36})

(implementer IPrintMyself { print-myself: (=> (print $this))})

(print (impl? $o IPrintMyself)) ; False

(sod o (mixin IPrintMyself $o))

(o.print-myself)

(print (impl? $o IPrintMyself))) ; True

So far you might think this looks a lot like classical inheritance; and you can use an implementer like a class with pretty good success. Here’s one that’s almost exactly a ‘base class’:

(do

(impl Person {name: '', age: 0, address: ''})

(sod dan (mixin Person {name: 'Dan', age: 36}))

(print (json $dan)))

Notice in the JSON that this prints out, there’s an address field on dan. Notice also that because we had our own values for name and age that they weren’t replaced with the implementer’s values. Similarly if you wanted to roll you own IPrintMyself you could do this:

(do

(implementer IPrintMyself { print-myself: (=> (print $this))})

(sod o1 {

name: 'dan',

print-myself: (=> (print `My name is {this.name}`))})

(sod o2 {

name: 'bob',

print-myself: (=> s (print `param was: {$s}`))})

(print (IPrintMyself? $o1)) ; True

(print (IPrintMyself? $o2))) ; False

The first weird thing that might jump out at you here is that neither of these things actually, explicitly, implements IPrintMyself; there isn’t a single mixin keyword anywhere in this sample. However shiro uses “duck-typing” to determine if implementers are implemented, which is to say if the object “quacks”, it’s considered to match. o1 in this example has a print-myself method on it which takes no parameters, which is the definition of the IPrintMyself implementer, so it quacks. o2’s print-myself takes a parameter, which the one on the actual implementer doesn’t, so it doesn’t quack.

The next weird/cool thing that jumps out at you is that we’re not using the *impl?* predicate like you’d expect. Every time you make an Implementer you also get what’s called an auto-predicate -- in this case IPrintMyself?. This is particularly useful for parameter predicates and should generally be the default way you check for quacking. Only use *impl?* when you’re dealing with a dynamic check where you don’t know at code-time what implementer you’re checking for.

## Interacting with Nimue (the TCP/HTTP/Telnet server)

Nimue is shiro’s network server (the name is a hold-over from back when the language was called Merlyn). It’s a very fast and basic TCP/IP server with a few extra modes to handle the kinds of network protocols I wanted to use with shiro, namely HTTP and Telnet. The most common thing you’re likely to do with Nimue is HTTP stuff, which we’ll spend most of this section talking about, but it’s useful to understand at least one of the other modes first before delving into the “complicated” one.

First let’s have a look at the world’s simplest telnet chat server:

(telnet 4676 (sendAll `{$id} says "{$input}"%n`))

If you evaluate this in the REPL you’ll notice something is weird right away -- it just kind of sits there not giving you the prompt for more input. This is because shiro has gone into network server mode, and when it does that it sits and waits, sometimes for a very long time (hypothetically, forever!). If you’re using shIDE, you will see a message in the lower left that says “Evaluating…” which is your hint that shiro is still doing something. If you try to run any other scripts shIDE will complain about it.

Once you go into network server mode a bunch of interesting things happen. In no particular order,

1. The interpreter's main thread (the one that executes your Shiro) begins blocking. Nimue, a multithreaded network server component will begin listening, and as events occur which evaluate shiro they will be evaluated by the network server's thread pool. Don't worry, your shiro is always thread-safe.
2. A series of local, let-scoped variables will be created for shiro evaluated in the server's context. For telnet (and TCP), these are id (a guid-as-string uniquely identifying the socket which triggered the evaluation) and input (the full line-command sent to the server for telnet, or just whatever input was received on the TCP socket).
3. Several different keywords will become available for use, depending on the type of the server. In this telnet example, they are send, sendTo and sendAll (same as for TCP), in addition to stop. These are called ‘contextual keywords’ in Shiro.
4. The main thread doesn't go away -- all your code and variables are still there, and if the network-server ever executes a 'stop' keyword it will come right back. You can even return something from the network thread to the main thread by passing it as a parameter to stop. Here's a telnet server that can be stopped:  
     
    (telnet 4676   
    (if (= $input "quit")   
    (do (print "quitting")(stop "Quit as instructed"))  
    (print $input)))  
      
   If you telnet into this server and type anything it will print out in the Shiro window. If you send it quit then Nimue will stop listening and return "quit" to the main thread.

So now that the basics are out of the way (and if you’re one of the three people on Earth desperate to learn more about how to make a MUD or MUSH in shiro don’t worry there’s a chapter on the telnet command in full, agonizing detail later), we can get to why you’re all here… web services! In keeping with my approach so far, let’s start incredibly simple then ramp up really fast so it feels like I’m teaching you something instead of just babbling. We’ll start by proving that Nimue can understand HTTP:

(http 8676 (print request.url))

Now when you fire this up and point your web browser to <http://localhost:8676/some/url/whatever> you’ll notice that the obvious thing happens (that url prints in your shiro interpreter). You’ll also notice that the same thing that’s printed is returned to the browser. You already know why that is, of course, it’s because print evaluates to whatever it printed, and Nimue’s HTTP server mode responds to requests with whatever the list passed in the second parameter evaluates to.

You might also notice that your browser is probably asking for favicon.ico, but you either know why that is or you don’t and it’s not really important.

Anyway, web services usually return JSON, so let’s try doing that.

(http 8676

(content "application/json"

(json {name: "Dan", age: 35})))

content is a contextual keyword (like send and sendAll in the telnet example earlier) which can only be used in HTTP mode. There are a whole bunch of them, which we’ll be dealing with in full, agonizing detail later on, but will touch upon now. content, obviously, sets the content-type of the response.

This lovely “web service” returns static JSON no matter what URL you plug into it; in fact it ignores the URL and the request completely. You can GET or POST or PATCH to it and it will always just return that JSON. It’s a little dumb, but it’s very, very fast at least. Let’s add some routing to it, so that it has different ‘pages’.

(http 8676 (route

"getJson" (content "application/json"

(json {name: "Dan Larsen", age: 35}))

"quit" (stop)

"default" (status 404 "Endpoint not Found")))

; Check out:

; http://localhost:8676/getJson

; http://localhost:8676/pageDoesntExist

; http://localhost:8676/quit

The string part of the route in the example above doesn’t have to be a string, it can also be a lambda (which gets pretty cool if you give up completely on trying to do routing the normal way). You don’t *have* to use route, of course, you can use the properties of the request object and implement whatever you want, but for most applications route makes things nice and simple.

There’s a lot more stuff I want to teach you about HTTP mode, but for now there’s just one more example that really is the culmination of shiro’s design philosophy. When I set out years ago with a hazy idea that LISP would make a really good node.js, the code snippet I started with in my head looked a lot like this, and now it works:

(do

(sod data '(

{id: 1, name: "Dan", age: 35}

{id: 2, name: "Dhiraj", age: 28}))

(http 8676 (route

"api/people\*" (rest $data id)

"quit" (stop $data))))

; You can now GET/POST/PUT/DELETE on

; http://localhost:8676/api/people

; like (GET):

; http://localhost:8676/api/people/1

You now have a fully functional REST server (it even has PATCH, lol). If you don’t bother with the seed data it’s three lines of code. When the server quits it returns the data with all the changes that were made by any REST calls. With a little bit of code to save that state and some more code to inject it at the beginning you could actually use this as a full, persistent REST service if you wanted to.

One last thing before we go -- any time something is going to take a very long time in shiro you have the option to continue on with your program and wait for the long-running thing to finish. You do this using the await keyword,

(do

(print 1)

(await res (telnet 4676

(if (= $input "quit")

(do (stop $input))

(print $input))))

(print 2)

(print $res)

(print 'done'))

If you run this code it does something pretty interesting. It prints 1, opens a telnet server, prints 2, and then sits there until you quit the telnet server, at which point it will print ‘quit’ and then ‘done’. Do you see what’s happening? When you await something you tell shiro that it’s going to take a long time (maybe forever), but that its okay and you don’t really care right now if it’s done or not. You can then go on and do anything up until you try to access the value of the variable you’re awaiting (res, in this example), at which point shiro recognizes that it’s still waiting (if indeed it’s still waiting) and will pause your code until it can give you a value back.

Thus we can print the ‘2’ right away (even though without the await the 2 won’t print until after the telnet server quits), but as soon as we try to print the value of the variable we’re awaiting everything pauses until it becomes available.

You could use this for all kinds of nefarious evil (like opening multiple TCP/IP servers from the same shiro program), or for good -- like parallelizing long-running tasks, being able to interact with a running service or waiting for long, blocking things without hurting your application’s performance.

The thing you’re awaiting gets *an exact copy* of your symbol table, including functions and variables. This is a ***copy***, meaning anything the thing being awaited does to those variables and functions is lost when it returns. The only thing you “get out” of something you’re awaiting is whatever the list evaluates to. This is how we ensure that no matter how many things you’re waiting for, shiro is always thread-safe. Any results of your async-list should be returned from the list.

Now there’s another keyword, awaith (or hermeticAwait) which does the await without cloning the symbols. This means you won’t have access to any of your variables or functions, but it does allow us to get going a bit faster and more efficiently. It’s still not something that should be used willy-nilly as it will always incur a performance penalty, albeit a lesser one.

Alright, there’s a bajillion more things to learn in shiro, but this chapter was about getting up and running quickly, especially for people who already know how to program. We’ve only just started scratching the surface, but that’s what the rest of the book is for I guess.

# Learn your Dev Environments: shiro and shIDE

## shiro

asdasds

## shIDE

asdasds

## VS Code / Other

asdasds

# All the Keywords (ugh…)

Writing this was every bit as not-fun as you’d imagine it would be. You’re welcome.

## Normal

### + - / \* (Arithmetic Keywords)

Shiro uses a LISP-standard prefix notation for arithmetic, meaning the operator is the command/keyword and it will apply to all the parameters in the list. There is no “order of operations” with this syntax – code is executed as is standard with Shiro. A discussion of basic arithmetic (and thus a breakdown of these keywords) is beyond the scope of this guide.

Examples:

; All the following equal 4:

(+ 2 2)  
(- 6 2)  
(/ 8 2)  
(\* 2 2)  
(- (\* 3 3 3) 5)

### = != ! (Equality Testing Keywords)

Testing for equality (or lack of equality) or performing a boolean NOT operation are all pretty standard in Shiro, using the three keywords shown above. The examples below should be fairly self-explanatory

Examples:

(= 2 2) ; T

(= 2 2.5) ; F

(!= 2 (/ 4 2)) ; F

(= nil "nil") ; F

(! true) ; F

(! nil) ; T

(! 0) ; T

### > < >= <= (Comparison Keywords)

The normal comparison operators are all available in Shiro, using the standard (command parameter…) syntax. Hopefully the examples below are self-explanatory.

Examples:

(> 3 2) ; T  
 (>= 2 2) ; T  
 (< 10 50) ; T

### .

The *.* keywordis used to dereference inline objects -- it is very similar to how dots are used in conventional programming languages, so:

(. Person Address City)  
-would look like this in a normal language:-  
Person.Address.City

. requires the properties you ask for to be present on the object -- if any of them are not an error will be thrown. If you are not sure the properties will be there, you can use the *.?* keyword instead, which is just like dot, but will return nil if any of the properties are not found instead of throwing an error.

Example:

(do  
 (sod o {name: "Dan", address: { street: "123 Main St",   
 city: "Portland"}})  
 (print (. $o address city)))

### .?

The *.?* keywordis used to dereference inline objects -- it is very similar to the *.* keyword except that where *.* will throw an error if any of the properties you ask for are not found, *.?* will return nil instead.

Example:

(do  
 (sod o {name: "Dan", address: { street: "123 Main St",   
 city: "Portland"}})  
 (print (. $o address city)) ; Portland  
 (print (.? $o adres city))) ; nil

### .call / .c

The *.call* keywordis used to call object-lambdas. Similar to keywords like *interpolate* and *impl?* it exists to serve a niche purpose in certain, obscure situations, but is mostly supplanted by the dot-unwinding reader shortcut. In almost any case where you’re calling an object lambda you can just dereference it as the command of a list, but every now and then you will need or want to explicitly call a lambda.

Example:

(do

(sod o {

name: "Dan",

introduce-myself: (=> (print `I’m {this.name}`)),

say-hi: (=> s (print `Hello {$s}`))

})

(.call $o introduce-myself ())

(.c $o say-hi ("steven")) ; is the same as:

(o.say-hi "bob"))

### .s .set / .d .def / .sod

This family of keywords is used to set and define properties on objects. They work just like their “undotted” counterparts except that instead of a single name they have a dereference list, just like with the *.* keyword. The final parameter to these keywords is the value to set the resulting property to.

Example:

(do

(sod o {name: 'steve' })

(print o.name) ; steve

(.sod o name 'dan')

(print o.name)) ; dan

### apply

The *apply* keyword takes two parameters, the first is a lambda, function name or keyword and the second is a list. Each item in the list will be evaluated with the command passed as the first parameter and the result will be inserted into a new list, which will be the ultimate result of the *apply*. This is different from *map*, which works similarly but returns the original, unaltered list.

Examples:

(do  
 (print (apply (=> (x) (+ $x 1)) '(1 2 3 4))) ; '(2 3 4 5)   
 (print (map (=> (x) (+ $x 1)) '(1 2 3 4)))) ; '(1 2 3 4)

### await / awaith

The *await* and *awaith* (short for hermetic-await) keywords are used to evaluate a list asynchronously so that your main interpreter can continue on doing stuff while some long-running thing happens in the background. Your main thread will only start blocking (waiting) if you try to access the value you’re awaiting the result of. *await* builds a complete copy of your local symbol and function tables for the new thread to use, which is a computationally- and memory-expensive process. *awaith* starts the thread with a completely fresh symbol table, which is more performant but can cause problems depending on your design.

Example:

(do

(print 1)

(await res (telnet 4676

(if (= $input "quit")

(do (stop $input))

(print $input))))

(print 2)

(print $res) ; blocking starts here

(print 'done'))

### concat

The *concat* keywordis used to concatenate multiple lists together into one (or to add single items to a list).

Example:

(do  
 (sod l '(1 2 3))  
 (sod l (concat $l '(4 5) 6 7))  
 (print $l))

### catch / throw

*throw* and *catch* represent one of Shiro’s two forms of structured exception handling (with the other one, which is more broadly applicable, being *try*). *throw* raises an exception, which of course in Shiro is just a list that you “throw”. If nothing catches this exception it will make it all the way up to the interpreter and will error out, so in the simplest form, *throw* is a way to define your own runtime errors.

However you can also *catch* things you throw farther up the list hierarchy. This allows you to handle errors and continue, report them gracefully, retry, etc.

*catch* **will not catch** anything that’s not thrown. If you have a syntax error or an invalid evaluation error which creates a normal Shiro error, catch will ignore it completely. Use *try* if you watch to handle both thrown exceptions and normal Shiro errors.

Example:

(do

(catch (throw "Dan was here")

(print "Exception was thrown 1")

(print "This also happens"))

(catch (throw "Dan was here") (print `Exception was thrown {$ex}`))

(catch (+ 2 2) (print "This won't print"))

(catch ((+ 2 2)(+ 3 4)) (print "This won't print despite the syntax error")))

While I haven’t been explaining most of the examples in this section, this one shows of quite a few things so let’s review it quickly. The first *catch* shows the final-list, which is an optional, final list that can be passed to catch which will **always** be evaluated, whether or not anything was thrown. The second shows a more conventional *catch* without the final-list; it also shows that the thing that was thrown is available in a local let scope in a variable named ‘ex’. The third one shows that nothing happens when nothing is thrown, and the final one whose that a syntax error (in this case a sibling-peered list) **will not**trigger the catch.

### contains

*contains* takes two parameters. The first is a list or string, the second is a value to search the first parameter for. It will evaluate to true if the first parameter contains the second and false otherwise.

Example:

(do

(print (contains "hello nurse" "hello")) ; T

(print (contains '(1 2 3 4 5) 3))) ; T

### def

*def* is used to define global variables. *def* should only be used the first time the variable is created, trying to *def* a variable that is already defined will cause an error, you should use the *set* keyword in this situation. If you don't particularly care to use *set* and *def* in this fashion you can use the *sod* keyword, which is short for SetOrDefine. *def* takes two parameters, a name and a value.

Example:

(do  
 (def x 1)  
 (def s "Hello world")  
 (def o {prop: 1}))

### defn

*defn* defines a named function at global scope. The resulting function can then be called anywhere in your Shiro code as if it were a keyword or library function. The first parameter to *defn* is the name of the function, the next is a list of parameter names (these parameters will exist as an implicit let-scope within the function body). Even if a function takes no parameters (as in the hello-world example shown below), an empty list must be passed here. This list has an implicit-quote as well, meaning it will not be evaluated. The third and final parameter to *defn* is the body of the function, generally wrapped within *do* block.

Example:

(do  
 (defn hello-world () (print "Hello nurse!"))  
 (hello-world)  
 (defn say-hello (name) (print (str "Hello " $name)))  
 (say-hello "Dhiraj"))

### def?

*def?* is a predicate which takes a name as a single parameter, it will return true if that name is defined is any of the current scopes which are accessible (including let-scopes). While most other, similar predicates (like *num?* or *list?*) take values, *def?* takes a name, so you do not use the reader-shortcut $ as you would in other cases.

Example:

(do  
 (sod l '(1 2 3))  
 (sod i 2)  
 (sod d 10.25)  
 (sod o {name: "dan", age: 35})  
 (sod s "Hello nurse")  
 (def? l) (def? s) ; T  
 (def? bob) ; F

### dejson

*dejson* performs the opposite operation that the *json* and *jsonv* keywords perform, attempting to turn JSON into a valid Shiro inline-object. JSON arrays will be processed into Shiro lists.

Example:

(do  
 (sod json '{"name": "Dan", "age": "35", "someShiro": ["str", "Hello ", "world"]}')  
 (sod obj (dejson $json))  
 (print (. $obj name)) ; Dan  
 (print (eval (. $obj someShiro)))) ; Hello world

### do

A value bit of Shiro code is always a single list at the top level -- similar to the first element of a tree. If you try to evaluate sibling lists (multiple lists which sit next to each other at the same level of the source tree) you will receive an error. The *do* keyword takes any number of lists as arguments, and evaluates them all one after another. It is generally used as the very first statement in your code, and also in areas which take blocks of code (like function body definitions)

Example:

;Try this without the do to see the error  
 (do  
 (print "First list")  
 (print "Second list"))

### enclose

The *enclose* keyword is used to create “private” variables to a particular object. It takes two parameters, both of which should be objects. The former set is the ‘enclosure’, the set of private variables and methods which will **only** be available within an object-lambda on that object. My least favorite keyword in the whole language, but if you absolutely have to hide stuff this is how you do it.

Note that in the following example, any attempt to reference o.\_p will result in a Shiro runtime error. You can only get at \_p if you’re evaluating a lambda which it itself attached to o.

Example:

(do

(sod o (enclose

{\_p: 1}

{getP: (=> (this.\_p)), incP: (=> (.sod this \_p (+ this.\_p 1)))}))

(print o.getP) ; 1

(o.incP)

(print o.getP) ; 2

(o.incP)

(print o.getP)) ; 3

### eval

Evaluates the list which is passed as its only parameter. *eval* is usually used in conjunction with *quote*d lists, or lists which are assembled programmatically.

Example:

(eval '(print "hello nurse"))

### filter

The *filter* keyword takes two parameters -- the first is a predicate (a keyword or function which ends in ? and returns a boolean) and the second is a list. *filter* will evaluate each item in the list against the predicate and return a new list with only those items which pass the predicate.

Example:

(do  
 (sod x '(1 2 "dan was here" "hello world"))  
 (print (filter str? $x)) ; dan was here hello world

### fn / =>

*fn* creates a lambda, or anonymous function, which can be assigned to a variable, passed as a parameter or invoked immediately. Lambdas can be used in many places in Shiro, including as predicates, as methods on objects and implementers and as parameters to other functions or lambdas. In the example below we are creating a lambda then invoking it immediately, then printing the result.

Example:

(print ((fn (x y) (+ $x $y)) 2 2))

### fn?

*fn?* is a predicate which takes a name as a single parameter, it will return true if that name is defined and references a function. While most other, similar predicates (like *num?* or *list?*) take values, *fn?* takes a name, so you do not use the reader-shortcut $ as you would in other cases.

Example:

(do  
 (sod l '(1 2 3))  
 (sod i 2)  
 (defn hello-world () (print "Hello nurse!"))  
 (fn? hello-world) ; T  
 (fn? l) (fn? i) ; F

### gv

*gv* is used to get the value of a global-scoped variable. In general, *v* or the reader-shortcut $ should be used to get variable values, *gv* should only be used when in a let-scope and attempting to get a global variable which may have been overidden.

Examples:

(do  
 (sod x 123)  
 (let (x 555) (  
 (print (gv x))))

### http

*http* causes Shiro to begin handling HTTP requests in a fashion similar to node.js. The main thread will block and only act when a request is received; the network server itself is multi-threaded and fairly performant (within reason), but your Shiro will always be thread-safe for you automatically and you don’t have to worry about multi-threading problems.

*http* takes two parameters, the first is a port number, and the second is the list that will be evaluated any time an http request is received. That list’s evaluation will occur within a special let-scope, which includes a variable called *request*, which contains the following properties:

**body:** The payload within the request (usually a POST body or something similar)

**url:** The URL that was requested (minus the domain name)

**args:**An inline-object containing name-value pairs of every query string parameter

**method:** The http method that was used for the request (GET, POST, etc.)

Whatever the list ultimately evaluates to will be returned as a result of the request. You can use certain, special contextual keywords (like *status* and *content*) to specify certain parameters of the response, like the HTTP status code and the content-type header.

Example:

(http 8676 (route

"getJson" (content "application/json" (json {name: "Dan Larsen", age: 35}))

"quit" (stop)  
 "default" (status 404 "Endpoint not Found")))

### if

As in most programming languages, the *if* keyword is used to execute code conditionally – that is, to only execute code if a particular condition is true or false. *if* can take either two or three parameters – the first is the condition to check, the second is the code that will only be executed if the condition is true, and the optional third parameter is a block of code to be executed if the condition is false (generally known as an ‘else’ in other languages).

In Shiro, almost every command returns something (usually one of the things passed to it, if it is not a transformational keyword), and *if* is no exception – if returns the result of the list is processed (if it processed one), or nil if no code was evaluated. This allows you to use *if* as a ternary operator (? : in most languages), as shown in the second example below.

Shiro uses a system for determining “truthiness” similar to JavaScript – which is to say it coerces things into booleans, treating non-zero numbers, lists which contain elements, strings with characters in them and inline-objects as true, and other things (including nil) as false.

Example:

(do

(if true (print "Hello world"))

(if false (print "Won't Print") (print "Will Print"))

; if acts like a ternary operator (? :) as well:

(if true 1 2) ; 1

(if false 1 2) ; 2

(if 0 1)) ; nil

### implementer / impl

The *impl* keyword is used to define an implementer, a somewhat wonky Shiro construct that is kind of like a class, kind of like an interface, and kind of like a plugin (see the various sections and chapters where we deal with OO in Shiro for a more extensive guide to implementers). The first parameter to *impl* is the name of the implementer being created, the second is an inline object which defines the implementer.

Example:

(impl IDoStuff { s:'string', n:123, f: (=> (s:str?) `{$s}-{$s}`)})

### impl?

The *impl?* predicate checks to see if a particular object “quacks”, which is to say if it is capable of being treated as a certain implementer. The first parameter is the value to be checked, and the second is the name of the implementer. Note that creating an implementer also creates an auto-predicate for that implementer (in the format <name of implementer>?) so you rarely use this predicate directly unless you don’t know the name of the implementer you’re checking for at code-time.

Example:

(do

(sod o {name: "Dan", age: 36})

(implementer IPrintMyself { print-myself: (=> (print $this))})

(print (impl? $o IPrintMyself)) ; False

(sod o (mixin IPrintMyself $o))

(o.print-myself)

(print (impl? $o IPrintMyself))) ; True

### import

The *import* keyword is used to load a DLL which contains Shiro libraries into memory. Once loaded, the libraries within the DLL can be referenced using the *use* keyword. *import* takes a single parameter, which can be either the string 'MSL' to load the Shiro Standard Library, or the file path to the DLL you are trying to load. If no path is specified Shiro will attempt to load it from the current directory.

Example:

(do  
 (import math))

### interpolate

A very rarely-used keyword (because there’s a much more legible reader shortcut for it), *interpolate* is the long-winded way of doing a string interpolation. The single parameter will be evaluated as a string and then processed for inline code (which is marked with curly brackets within the string). Check out the string interpolation reader shortcut for the way you’ll want to achieve this effect 99% of the time.

Example:

(print (interpolate "2 + 2 is {+ 2 2}"))

### json / jsonv

The *json* keyword attempts to create legal JSON from the Shiro data passed as its first and only parameter. The most common use is to pass it an inline object, but it will turn any Shiro list into a JSON array if possible. *json* will not evaluate any Shiro code it encounters in the inline object, it will simply attempt to JSONify it (effectively recursively applying the *json* keyword to all children); if you wish to evaluate Shiro within this context you can use the *jsonv* keyword.

The "opposite" of this keyword is *dejson*, which turns valid JSON into a Shiro list which can then be interacted with in the normal fashion

Example:

(do  
 (sod x {name: "Dan", age: 35, someShiro: (str 'Hello ' 'world')})  
 (print (str "json: " (json $x)))  
 (print (str "jsonv: " (jsonv $x))))

;Output:  
 ; json: {"name": "Dan", "age": "35", "someShiro": ["str", "Hello ", "world"]}  
 ; jsonv: {"name": "Dan", "age": "35", "someShiro": "Hello world"}

### kw

The *kw* keywordreturns the first item (also known as the ‘command’) in the list passed as it’s parameter. If a non-list is passed, that value will be returned by itself without transformation

Example:

(kw '(1 2 3)) ; 1

### len

The *len* keyword evaluates to the length of the string or list passed as its only parameter.

Examples:

(len 'Hello World') ; 11

(len '(1 2 3 4 5)) ; 5

### let

By default the variables you *set* or define (or *sod*) are global variables, they exist throughout your Shiro code and are accessible anywhere. To create "local variables" you can use the *let* keyword, which creates a temporary scope level containing the variables you declare in the *let*. Variable in a 'let-scope' like this will trump global variables -- so if you have global variable x and a variable named x in your let-scope, getting the value of x with (v x) or $x will return the value of the let-scoped variable. Once the let-scope goes away, the global variable will still be there with its old value.

From within a let-scope you can access the value of a global variable with the *gv* keyword. The first parameter to *let* is a rare example of a list with an implicit quote in Shiro, meaning it will not be evaluated. It expects any even number of parameters, being name-value pairs representing the variables to be created and their values. Note that you cannot *set* or *sod* a let-scoped variable, as these keywords interact with the global scope level. You can instead *relet* a let-scoped variable to change its value.

Example:

(do  
 (sod x 123)  
 (print $x) ; 123  
 (let (x 555) (do  
 (print $x) ; 555  
 (print (gv x))))) ; 123

### list?

*list?* is a predicate which takes a single parameter, it will return true if that parameter evaluates to a list. Note that inline-objects of any sort are considered lists and can be treated as lists anywhere in Shiro, so they will be considered lists for the sake of this predicate.

Example:

(do  
 (sod l '(1 2 3))  
 (sod i 2)  
 (sod d 10.25)  
 (sod o {name: "dan", age: 35})  
 (sod s "Hello nurse")  
   
 (list? $l) (list? $o) ; T  
 (list? $i) (list? $s) ; F

### lower / upper

*lower* and *upper* return the lowercased and uppercased version of the string passed as their only parameter.

Example:

(do

(sod s "Hello Nurse")

(print `upper: {(upper $s)}, lower: {(lower $s)}`))

### map

The *map* keyword takes two parameters, the first is a function name, lambda or keyword and the second is a list. Each item in the list will be evaluated with the command passed as the first parameter. unlike *apply*, *map* returns the original list unaltered and ignores whatever the command passed in evaluates to.

Examples:

(do  
 (defn say-hi (name) (print (str "Hello " $name)))  
 (map say-hi '("Dan" "Dhiraj" "Dave"))

### mixin

The *mixin* keyword allows you to munge one or more implementers into an object. This can serve a variety of purposes depending on how you’re interacting with OOP in Shiro, from “instancing” a base class to implementing an interface to plugging in a piece of functionality. *mixin* takes at least two parameters; the last one is the object having stuff mixed into it, and the rest of them are the implementers to mix in.

Example:

(do

(implementer IPrintMyself { print-myself: (=> (print $this))})

(sod o (mixin IPrintMyself { name: "Dan", age: 36}))

(o.print-myself))

### nil?

*nil?* is a predicate which takes a single parameter, it will return true if that parameter evaluates to nil.

Example:

(do  
 (sod o {name: "dan", age: 35})  
  
 (nil? (.? $o fakeProperty)) ; T  
 (nil? $o) (nil? (.? $o name))) ; F

### nop

The *nop* keyword takes any number of parameters and does nothing with them, except letting you know that a NOP was evaluated. It generally it not useful in final programs, but can be a useful keyword if you want to temporarily exclude part of a Shiro program from evaluating (akin to "commenting out" some code in other langauges). *nop* always lets you know it was evaluated, if you would like to quietly skip some code, you can use *qnop* (quiet-nop) instead. *nop* always returns nil.

Example:

(nop (print 'hello world'))

### nth

*nth* returns the Nth parameter of a list, where N is the first parameter passed into it, and the list is the second. Indices in Shiro are 1-based, so the command/first parameter of a list is 1.

Example:

(nth 2 '(1 2 3)) ; 2

### num?

*num?* is a predicate which takes a single parameter, it will return true if that parameter evaluates to a numeric value, and false otherwise. *num?* will accept either decimal or whole-integer values as numeric.

Example:

(do  
 (sod l '(1 2 3))  
 (sod i 2)  
 (sod d 10.25)  
 (sod o {name: "dan", age: 35})  
 (sod s "Hello nurse")  
   
 (num? $i) (num? $d) ; T  
 (num? $l) (num? $o) (num? $s)) ; F

### obj?

*obj?* is a predicate which takes a single parameter, it will return true if that parameter evaluates to an inline-object, or any list which has named properties in it.

Example:

(do  
 (sod l '(1 2 3))  
 (sod i 2)  
 (sod d 10.25)  
 (sod o {name: "dan", age: 35})  
 (sod s "Hello nurse")  
   
 (obj? $o) ; T  
 (obj? $l) (obj? $i) (obj? $s) ; F

### pair

*pair* is used to create a named element, often used in quoted data lists or when appending properties to inline objects. It takes two parameters, a name and a value. You can visualize the end result of *pair* as a single-property inline object, like so:

(pair name value)  
-is-  
{name: value}

There are, however, a few things pair can do that inline objects cannot -- the name can be a list we are evaluating and not just a constant value. It can also be numeric, or something which would result in invalid JSON (for example, a string containing spaces). The resulting object might have problems serializing to JSON though.

Example:

(do  
 (sod o {name: "Dan"})  
 (set o (concat $o (pair age 35)))  
 (print (. $o age))) ; 35

### params

The *params* keywordreturns the parameters (all of the values except the first one) from the list passed as its parameter. If the parameter supplied is not a list, *rest* returns nil.

Example:

(rest '(1 2 3)) ; 2 3

### print

It will probably not surprise you to learn that the *print* keyword will print out whatever you pass to it to the console or output log (depending on which implementation of Shiro you are using). It can take any number of parameters and it will print them each out on their own lines.

Example:

(print "Hello Nurse!")

### printnb / pnb

Works just like print, but without printing any line breaks (thus the nb, or ‘no break’).

Example:

(pnb "Hello there " "Dan...")

### qnop

The q*nop* keyword takes any number of parameters and does nothing with them. It generally it not useful in final programs, but can be a useful keyword if you want to temporarily exclude part of a Shiro program from evaluating (akin to "commenting out" some code in other langauges). *qnop* does nothing quietly -- if you want a reminder than a NOP was evaluated in your code you can use *nop* instead, which will always inform you that it was evaluated.

Example:

(qnop (print 'hello world'))

### quote / '

In most circumstances when Shiro encounters a list, it will attempt to evaluate it, which means look at the first item in the list and treat it as a keyword or function call. You will sometimes want lists to be treated as data though, which is where the *quote* keyword comes in. *quote* tells Shiro not to try to evaluate the list that follows, and instead to simply treat it as data. Shiro's reader has a shortcut for quote, which is simply to put a single quote at the beginning of the list outside the parenthesis. Some lists in Shiro have an "implicit quote", which is to say the list is not evaluated when it is encountered, but you only find this in reference to specific parameters of certain keywords.

Example:

(do  
 (print (str "Hello " "world")) ; Hello world  
 (print '(str "Hello " "world")) ; str Hello world  
 (print (quote str "Hello " "world"))) ; " " "

### range

*range* returns the a sub-list containing a range of parameters from the target list. The first parameter to *range* is the 1-based index that we start from. The second parameter is the number of items to pull from the list, and the third parameter is the list itself. If the number of items requested is more than are available in the list then we will stop at the end of the list.

Example:

(range 2 2 '(1 2 3 4)) ; 2 3  
 (range 2 100 '(1 2 3 4)) ; 2 3 4

### relet

The *relet* keyword is used to change the value of an existing let-scoped variable. The variable must exist within a current let-scope, or an error will be thrown.

Example:

(do  
 (sod x 123)  
 (let (x 555) (do  
 (print $x) ; 555  
 (relet x 987)  
 (print $x)))) ; 987

### set

*set* is used to change the value of an existing global variable. The variable must have already been defined, using either the *def* keyword, or *sod*, which does double duty as either *set* or *def* depending on context.

Example:

(do  
 (def x 1)  
 (set x 2)  
 (set x "Types are highly overrated"))

### skw

*skw* is short for 'Set KeyWord' and is used to set the first value of a given list (which in most cases is the keyword) to a particular value. It doesn't replace the value in the list currently, but rather prepends the value you specify onto the list. You can use this either to prepend data to lists or to build dynamic lists for evaluation. The *eval* keyword can be used to evaluate the resulting list.

Example:

(do  
 (sod l '(1 2 3 4))  
 (eval (skw print $l)))

### sod

The *sod* keyword is short for 'set or define', and can stand in for either the *set* or *def* keywords. If you don't really care whether or not you are creating the variable for the first time or not you can use *sod* to guarantee that it will work, and in general use *sod* should be your goto keyword for declaring global variables.

Example:

(do  
 (sod l '(1 2 3 4))  
 (sod x 2)  
 (sod x "Types are highly overrated"))

### split

The *split* keyword slices a string into a list using a particular delimiter as a marker. It is most commonly used to break up strings into their composite words.

Example:

(print (nth 2 (split "A list of words is a string" " "))) ; list

### str / string

The *str* keyword simply concatenates all the values passed as parameters into a single string.

Example:

(do  
 (sod name "Dan")  
 (print (str "Hello " $name ", nice to meet you")))

### str?

*str?* is a predicate which takes a single parameter, it will return true if that parameter evaluates to a string. Note that the parameter must presently be a string – anything in Shiro can be treated as a string for display/IO purposes, but they will not pass this predicate unless they are already in string form.

Example:

(do  
 (sod l '(1 2 3))  
 (sod i 2)  
 (sod d 10.25)  
 (sod o {name: "dan", age: 35})  
 (sod s "Hello nurse")  
   
 (str? $s) ; T  
 (str? $i) (str? $d) (str? $l) ; F

### telnet / tcp

The *telnet* or *tcp* keywords put Shiro into network server mode, where it will wait for connections on the port you specify in the first parameter, accept them and start an input-buffering loop. This server can handle multiple connections simultaneously (like a chat server or MUD/MUSH might) and will buffer all input on a connection until it receives a newline, which will cause it to process that command.

Within the telnet server context every connection is identified by a unique id, which is a string in Shiro and a GUID in C#. You can use this unique ID to attach other metadata to the connection as necessary for whatever application you are developing. *telnet* and *tcp* take up to three parameters (with two of them being required): the port number, a list which will be evaluated each time a command is received, and an optional list which will be evaluated each time there is a new connection is made.

Within the command-handler list, there are two let-scoped variables which Shiro will automatically create for you. *id* has the unique id of the connection that send the command, and *input* is a string which contains the command received. With the connect-handler list, the *id* variable is present, but *input* is not. In either context you can access a variable called *AllConnections* which is a list of every connection id that’s currently active.

The example shown below is for a very simple, anonymous chat server. It will listen for new connections on port 4676, and when input is received it will send it to everyone connected to the server. Sending the input ‘quit’ without quotes will cause the server to stop listening and close all open connections.

Example:

(telnet 4676   
(if (= $input "quit")

(do

(print "quitting")  
 (stop $input))

(sendAll (str $id " says '" $input "'")))))

### try

*try* is the more generic of Shiro’s two options for exception handling (with the other being *catch/throw*). *try* works almost exactly like *catch*, except it’s not only looking for things you *throw*, but also **any Shiro errors at all*.*** *try* then is a superset of *catch*.

Note that the only difference in this example vs. the one we used for *catch* is the 4th one, which triggers in the case of *try*, but no *catch* due to the sibling-peered list evaluation syntax error.

Example:

(do

(try (throw "Dan was here")

(print "Exception was thrown 1")

(print "This also happens"))

(try (throw "Dan was here") (print "Exception was thrown 2"))

(try (+ 2 2) (print "This won't print"))

(try ((+ 2 2)(+ 3 4)) (print "This prints (syntax error)")))

### v

*v* is used to get the value of a variable (either a global or local/let-scope variable). It will try to find the most locally-scoped variable that is named whatever you pass it -- so let-scope variables will be checked first, then global variables, and finally auto-variables, which are variables injected by whatever is hosting your Shiro interpreter or compiled code. Since using *v* every time you want to access a variable's value is cumbersome, there is a reader shortcut using the $ before a name. A visual example:

(v x)  
-is the same as-  
$x

You should rarely have to use the *v* keyword directly, instead favoring the reader shortcut shown above. *v* is used sometimes however when building dynamic lists for evaluation.

Examples:

(do  
 (sod x "Dan was here")  
 (print (v x)) ; ...is the same as:  
 (print $x))

### while

*while,* as in most programming languages, is used to continue looping until a certain condition is true. The first parameter is the list to evaluate for truthiness, and the second is the list to continue evaluating until the condition becomes true.

Examples:

(do

(sod x 10)

(while (> $x 0) (do

(print $x)

(set x (- $x 1)))))

## Contextual - Telnet / TCP

### send

The *send* keyword sends a string back to whatever telnet connection we are presently addressing – in the command handler list this is the connection that sent the command, and in the connect handler list it is the connection that triggered the event. It is a shortcut:

(send "Hello world")  
-is equivalent to-  
(sendTo $id "Hello world")

The example shown below is a simple telnet echo server – it returns whatever commands it receives to the socket which issued the command.

Example:

(telnet 4676

(send $input))

### sendAll

*sendAll* sends a given string to every open telnet connection, including the one which triggered the current handler. The example shown below is a simple, semi-anonymous chat server:

Example:

(telnet 4676   
 (sendAll   
 (str $id " says '" $input "'")))

### sendTo

The *sendTo* command takes two parameters, the first is a string containing the connection id that we want to send something to, and second containing a value (or list to be evaluated) with what we want to send. The example below is a simple echo server which uses *sendTo* instead of *send*.

Example:

(telnet 4676

(sendTo $id $input))

### stop

The *stop* keyword is available when Shiro is acting as a network server (either *http* or *telnet*). It tells Shiro to shut the server down, close any open connections, stop listening, and return whatever parameter you pass it (if any) back to the list that started the network server. The example shown below starts a telnet server, and returns the first command it receives back to the list that started the server.

Example:

(print (telnet 4676   
 (stop $input)))

## Contextual - HTTP

### content

The *content* keyword is used to set the HTTP Content Type header for the response. It takes two parameters – the first is the content-type that should be used, and the second is a list that will be evaluated to determine the return value of the request.

Example:

(http 8676

(content "application/json" (json {name: "Dan Larsen", age: 35})))

### rest

*rest* implements a fully-functional RESTful service endpoint. It will evaluate to the result of the REST operation, based on the HTTP method, and will modify an in-memory data set based on the REST transactions. Nimue’s REST mode supports: GET, PUT, POST, DELETE and PATCH. The first parameter to the *rest* keyword is the data set we will operate on, the second one is the name of the unique ID for the records.

Example:

(do

(sod data '(

{id: 1, name: "Dan", age: 35}

{id: 2, name: "Dhiraj", age: 28}))

(http 8676 (route

"api/people\*" (rest $data id)

"quit" (stop $data))))

### route

*route* is used to help you route requests based on the requested URL. While it is possible to route manually by looking at the request.url property, *route* makes it easier and makes the resulting code flow a little bit better. *route* can take a dynamic number of parameters, but that number must be even – forming a set of string:value pairs which represent endpoints and their handlers.

Example:

(http 8676 (route

"getJson" (content "application/json" (json {name: "Dan Larsen", age: 35}))

"quit" (stop)  
 "default" (status 404 "Endpoint not Found")))

### status

The *content* keyword is used to set the HTTP Content Type header for the response. It takes two parameters – the first is the content-type that should be used, and the second is a list that will be evaluated to determine the return value of the request.

Example:

(http 8676

(status 404 "Resource not found"))

### stop

The *stop* keyword is available when Shiro is acting as a network server (either *http* or *telnet*). It tells Shiro to shut the server down, close any open connections, stop listening, and return whatever parameter you pass it (if any) back to the list that started the network server. The example shown below starts a telnet server, and returns the first command it receives back to the list that started the server.

Example:

(print (http 4676   
 (stop `I got a request`)))

## Reader Shortcuts

Shiro provides a bunch of reader shortcuts -- special syntax-helpers which let you make the kinds of list you’ll be frequently dealing with quickly and without headache. Some of them exist for readability, like dot-unwinding, while others are just there to prevent list-bloat and make your code a bit shorter and terser (like string-interpolation and quoted lists).

You as the programmer don’t have to worry about them being ‘reader shortcuts’, you can just use these conventions in your code and everything else happens under the hood for you.

### Quoted Lists

This was the very first reader shortcut we encountered, it allows you to never have to use the *quote* keyword, and instead to represent value lists like this:

(sod l '(1 2 3 4))

Which in turn becomes the following Shiro at evaluation time:

(sod l (quote 1 2 3 4))

As a general rule with quoted-lists, be careful not to have them end up in a place where Shiro tries to evaluate them as values.

### Dot Unwinding

Shiro’s dot keyword (.) was an obvious place for a reader shortcut… fifteen minutes into writing tests for it and I already hated the LISP-style syntax for deep dereferencing, and so dot-unwinding was born. You can use it to access the value of any particular property of an object by dereferencing it just like you would in C# or JavaScript. You can **never** set values this way though. You can call functions/lambdas though. Let’s take a look.

(do

(sod o {name: 'Dan', say-hi: (=> (print `hello {this.name}`))})

(print o.name)

(o.say-hi))

So much easier! The perceptive reader will have noticed that there’s another reader shortcut in that example, namely...

### String Interpolation

Never having to call string.format in C# is one of my favorite things to come out of the language since lambdas, and so of course Shiro has built-in string interpolation. It’s normally done with the hideously-long *interpolate* keyword, followed by a string which has executable/interpolated blocks within it marked with curly braces (look at the ‘say-hi’ method in the code sample just above for an example).

Because no one wants to ever type interpolate, you can instead use ticks (these things: `, the one near the ~ on your keyboard) to mark your strings and the reader will automatically apply the interpolate keyword, like so:

print `2+2 = {+ 2 2}`

### AutoV

The reader shortcut you’re most likely to get confused by is this one, not because it does anything complex, but because it will take you a bit to get used to when you need a variable’s **name** in Shiro and when you need the variable’s **value**. It’s in the latter case that you use this shortcut, which is simply that putting a $ in front of a name wraps it in a (v) list. So an obvious example:

(do

(sod x 123)

(print $x))

We obviously don’t want to print the name ‘x’, we want to print the value of x, so we know we want a *v*, and thus use the AutoVar. Likewise if we’re setting a property on something we want the variable, not the value:

(.sod o name 'bob')

Finally the only case that’s slightly ambiguous is when we’re getting values off objects. It seems obvious (we want the value, not the variable), and so if you’re doing it normally you use the AutoVar. However if you’re using the dot-unwinding reader shortcut mentioned above you don’t. So these two lines of code are correct, and equivalent:

(print (. $o name))

(print o.name)

Every now and then even I screw it up and forget a $ or put one where I shouldn’t, but I’ve yet to find that it doesn’t make sense within the paradigm I described above when I do, it’s just that I chub something.

# The Shiro Standard Library (double ugh!)

And just when I thought I was done with the monotony…

## math

asdasds

## files

asdasds

# Idiomatic Shiro

## With Great Power Comes Great Responsibility...

Shiro wins every “how few lines of code can you do this in?” competition for a couple of reasons -- the main on is that it’s a very terse, high-level language that can do a lot with a few syntax elements. The other one is that you can technically write any shiro program as a single line of code -- it’s all just a single list after all. Just because you can write everything on one line and count on your IDE’s brace-matching to sort it all out though doesn’t mean you should.

Shiro is like that, a lot. You can do anything, and I do mean *anything*. There is no compiler to tell you no -- shiro’s compiler just shoves your source and the interpreter together into an executable and lets it sort itself out at runtime. The interpreter and the parser will do their very best to interpret anything you supply before giving up with an error. There isn’t even lexical analysis (meaning that other than rank basics like matching parenthesis we don’t even know if shiro is syntactically-valid until runtime; and of course since shiro evaluates lists and those lists can change at runtime they can always enter an invalid state later on).

Because of this great power you have to be mindful of how you write shiro and how you use it; it’s a lot like JavaScript in that way, except without the decade of tooling that’s evolved to make JavaScript safer. If you’re dashing something off for yourself that will be used twice then of course, go nuts, but the moment you’re considering a longer-lived shiro application you have to start thinking about it just like you would a ‘real’ project.

This section is about some of those concerns, things to think about for larger and more important projects, and suggestions in terms of project layout, file formatting and code conventions.

### How Are You Going to Write Your Shiro?

In a way the act of writing a larger Shiro project is about building up a vocabulary for yourself to address the problem domain. Done correctly you end up with something like Nimue’s http subsystem, where a single keyword can drive an entire, complex protocol-based workflow. Done incorrectly you end up with an unintelligible dialect which cannot be extended in the ways you need and doesn’t flow into idiomatic Shiro very well.

So the first thing to consider if what kind of vocabulary you want to work with. Do you want your top-level Shiro files to read a lot like code written in ordinary, OO languages? Do you want a more FP approach where you’re passing lambda-chains around? Maybe you just want a simple expansion of Shiro’s LispScript dialect to support a few higher-order functions? Or maybe you just want to go crazy and mix-and-match all the different ways to come up with one that meets your exact needs?

There is no “right” answer to this question, not even one approach which is considered “more idiomatic” than the others. If you’re writing a program that deals heavily with objects and their interactions and relationships with each other, write it OO. If you’re dealing with a bunch of atomic processes, consider FP. If you’ve got objects and atomic but variable processes, build a solid set of objects to represent your concretes and then use a procedural/functional dialect to interact with them.

The only wrong answer (for larger projects only) is, “Well I haven’t really thought about it,” because what you’ll end up with is a bunch of discrete, detached functions and objects that don’t flow together well, don’t function in a Shiro-like way and don’t play nicely together. The resulting code will be bloated, hard to read, impossible to maintain and difficult to expand.

Once you’ve settled on a general programming paradigm (or set thereof) to use, the next question you need to ask yourself is: do you feel lucky, punk?...

### Deciding How Much Safety You Want

Shiro does not force any sort of safety on you that you don’t want; this is by design. You can happily try to dereference properties off strings or evaluate completely nonsense lists, or pass strings to functions expecting objects, or whatever you want. Most of the time this will result in a helpful error message pointing you at exactly where the problem is (or as close as we can tell -- sometimes evaluating Shiro gets weird), but that doesn’t really help you very much when you’re dealing with an important service and it’s Saturday at 3 AM and you’re getting an error saying that a list you never deliberately tried to evaluate can’t evaluate.

Shiro’s parameter-safety mechanisms exist to help mitigate these problems, detect them early, and provide more sensible and obvious error messages when the prerequisite conditions are violated. The single most useful and common of Shiro’s safety mechanisms is the predicate-parameter, which is where you attach a predicate to the parameter of a function of lambda. This tells the interpreter that -- whenever we call that lambda or function -- the passed value has to pass the predicate for the call to be valid. You will get a very specific error message when this doesn’t happen with all the information necessary to figure out what’s going on.

So if I’ve got any ivory-tower architects still reading this thing (and I doubt it, I figure I lost them ages ago), I can already feel them deciding to *always* use predicate-parameters, because why wouldn’t you? Turn Shiro into TypeShiro with this one, easy hack! The performance hit is pretty negligible after all...

And I mean, you can. I personally feel you’re going way overboard, but you can predicate every single parameter. You can even write a bunch of custom predicates and get really, really (, really) specific with your pre-call checks. Sometimes this can even be cool (like doing object validation automatically as part of a predicate-parameter).

Just be advised you’re going to be annoying some people downstream. Not every function *has* to take a parameter of a certain type. Sometimes checking for a lower-level mixin is better than checking or a higher level one (check IPrintMyself instead of APerson for example, assuming APerson normally implements IPrintMyself).

Another area where “safety” can come up is with mixins and implementers. Are you going to enforce they *only* be used in a particular way? Are you okay with monkey-patching, or do you want all your implementers to represent either concrete objects or promises to implement functionality? Is the ability to hide things within your objects important? Are you sure you’re not going to want to unhide those things eventually?

The idea that Shiro is just Shiro (for better or worse), and that the individual developer or team kind of “opts in” to different kinds of safety (and perhaps “opts out” of certain possible uses of the language). Have these discussions up front so that one guy doesn’t build a custom list-building factory that generates code at runtime to evaluate while the other guy is writing type-safe, classical OO code.

### What Goes Where - Shiro Projects

asdasd

## Style and Structure

There are some unique considerations when writing in LISP syntax (and Shiro specifically) which may be unfamiliar to you, and even a bit unintuitive at first clip. This section covers them.

### Bury your Branches

In most programming languages your branching instructions are usually on the outermost layer. It’s not unusual for a function to be some setup, then a couple of nested if-then-elses which represent the logic of the function. In shiro we write the same sort of blocks, but the conditions should go as deeply as possible, not at the top. For example, it's always better to write:

(print (if $whatever 1 2))

Then

(if $whatever (print 1) (print 2))

Because everything in shiro evaluates to something, you can often structure your code in much more efficient and pleasant-to-read ways than you might expect. Whereas in C# or Java I often look to the outermost scope levels of a function to figure out the general “shape” of it and what it does, in well-written shiro it’s often best to look at the deeper elements and work your way up and out.

This applies most obviously to if statements, but it also applies to looping operations (or map/apply operations in shiro) and whatnot. In idiomatic code, often the outermost layers of the function are the *functional* ones which actually describe the mechanics of what’s happening, with the innermost levels dealing with sanity checking, filtering and control flow.

If you find yourself often having to structure your code in a more conventional format, like in the second example above, it’s probably because you’re failing to…

### Compose Lambdas for Patterns

The ability of a lambda value to be a command is a key part of shiro that you’ll use sometimes without really thinking about and can often take for granted. To remember what I’m talking about, let’s go back to the world’s most obtuse way to calculate 2+2:

((=> (x y) (+ $x $y)) 2 2)

Shiro evaluates from the inside out, so the first thing it does it build that lambda (which takes two parameters, x and y, and adds them together). Now the outer list which it tries to evaluate is:

(<that lambda we just made> 2 2)

Which is a pretty obvious line of code once you get past the fact that the command is an in-memory data structure representing a lambda and not anything you can type out. You can use this feature, especially in combination with the previous advice, to move things around in shiro code to where you want them. Your if statements can easily return lambdas which are evaluated or passed in to map/apply/filter, and things like that.

Basically any time there is a packet of functionality you want to perform more than once, make it a lambda or a function. The barrier to creating a new function/method in your static language of choice is higher than in shiro, and so you should almost *never* copy and paste code in shiro. Rather, package the code in a way that it can be used in multiple places.

When you build up an intelligent set of functions and lambdas for your project you find your shiro ‘vocabulary’ becomes increasingly high level until you can flit about your problem domain with very, very few commands and lines of code. Think of the contextual-keywords in Nimue, something like ‘rest’, which can invoke a whole complex REST server with a single keyword. Shiro is a compositing language where you can build yourself a whole development environment specialized at solving whatever problems you’re trying to solve.

So get in the habit of making lambdas, and writing code that takes lambdas as arguments. Any time you’re reproducing a code list twice in the same file it indicates a structural problem in your architecture take a step back and figure out where you can make it more efficient.

### Code From the Inside Out

When I’m coding something in C# my cycle almost always goes from the outside in. I make the skeleton of a function, call it, build and test. Then I flesh out that function to the next one, stub out the next one, build and test. Etc. Etc. When I’m coding in Shiro I do exactly the opposite. I start with innermost, specific bits of functionality and build onto them, wrapping them in more and more lists as I build towards my goal. It’s different and it takes some getting used to, but it’s much faster and easier once you get the flow of it.

Part of that is just LISP syntax. It’s easier to wrap a list in a list than to insert a list into a list, both in your editor and in your “mental map” of your code. Going deeper into Shiro can feel like breaking apart what you’ve already built, while building outwards maintains all the stuff you made earlier and expands on it.

If there’s a theme to these suggestions so far, it’s basically that Shiro is backwards, and this suggestion is the embodiment of that idea. The more you become comfortable with the syntax and work with the language the more you’ll find yourself thinking in reverse from the way you do when writing those Algol-based, curly-brace languages.

### Use let-scopes and Namespace Objects

Don’t stink up the global symbol table with what are really local, short-term variables. If you need a variable for a particular task (and only for that task), put it in a let-scope. Just because Shiro doesn’t bother making a new scope level every time you fart or type a parenthesis doesn’t mean that you should just embrace having everything be global. Shiro lets you define you own scope levels cheaply precisely so that you can put them *exactly* where you need them.

This has a lot of positives. The obvious one is cleanliness in your global variables, which can be much, much more important than you might think. If everyone uses ‘i’ as a loop counter in their code in Shiro, and if they’re using the global ‘i’, then if you call a function that loops inside a loop of your own you’re going to bork your own ‘i’. Now of course we don’t really use loop-counters very often in Shiro, but you get the idea. The solution to this is to make your own ‘i’ and not mess with the global one, if one exists.

Another benefit is faster awaiting. The less junk you have in the symbol table, the faster we can spin up a new interpreter for await (this doesn’t apply to awaith of course).

Another thing to think about is name collision. Most libraries and files in Shiro have some settings and actual global variables they want to keep around. Rather than directly naming them and risking a name collision with another library or file, it’s best to use what I call namespace objects -- special inline-objects which contain your settings, like this:

(do

(def myFileSettings { setting1: 123, setting2: "blah blah"})

...)

Using def instead of sod gives us a measure of runtime safety in case someone else tries to define another one of these over the top of us -- although it will cause problems if we include our library twice. You can use def? to check and raise an error that way instead.

### Indentation and Blank Lines

Indentation and brace-matching are less-useful in LispScripts than in other languages (although they are still important). Things tend not to match up at a scan, because you’re often closing a *lot* of lists on a single line. In C# or JavaScript you can scan down the indentation to scope levels in well-formatted code. In shiro you can… kind of… sort of do that a little bit. I mean technically if you wanted to write ugly shiro you could code this way:

(await res

(telnet 4676

(if

(= $input "quit")

(do

(stop $input)

)

(print $input)

)

)

)

(print 2)

Which lets you scan down the lists with the extra end-parentheses on their own lines. I personally *hate* this style, but hey you downloaded my programming language you can write it however you want. In my opinion, this is the correct formatting for that code:

(await res (telnet 4676

(if (= $input "quit")

(do (stop $input))

(print $input))))

(print 2)

Which maintains the important levels of the list (await -> if -> then) without wasting lines or indentation. Notice how the do is combined with its first parameter; even though it’s technically a new ‘list-level’ it’s not an important one, so we don’t mark it with an indentation. Similarly the telnet is “attached” to await in a logical way so it doesn’t need its own indentation level.

Shiro lists go *MUCH* deeper than blocks in JavaScript and C# and similar languages (just imagine if every single keyword in C# was followed by a block). You can’t give each list its own indentation level or you’re going to make your code ugly and unreadable. Figure out how to group lists into logical blocks and indent them based on functionality and their place in the larger flow of the program.

The effect of what I just described is that shiro code is *dense*, quite literally. If you compare the same program written in shiro and C#, you’ll find that the C# version is taller and contains much more whitespace and lines with very little on them, while the shiro version is squatter and wider (not counting whitespace) and contains almost no superfluous space and lines. Especially if the shiro in question is idiomatically-written.

To mitigate that, blank lines become much more important. Whitespace lines are *always* important in programming, but in shiro they are your main tool to mark up breaks in your files, and they are completely under your control. Java and C++ kind of “force” you to structure your code into logical blocks unless you really, really don’t want to. Shiro is effectively structureless and so if you’re counting on a pretty-looking source file to emerge just from writing it… well, don’t.

### When to Await

You can await literally any list in shiro, anything at all. A print statement, a variable assignment, a no-op (an instruction that literally does nothing and evaluates to nil). They’re all awaitable.

Awaiting -- and thus multithreading -- are syntactically cheap in shiro. In fact, I'll go out on a limb and say that multi-threading in shiro is syntactically-cheaper than in any other programming language in human history. It’s also *safe*. There’s no locking, no race conditions to worry about… about the worst trouble you can get in is recursively awaiting something indefinitely and I had to think for like two whole minutes to even come up with a code snippet to reproduce that and test that it errors out correctly.

Because of this (once you get over the natural and healthy fear of threading that almost every programmer has) it can be easy to go crazy with it. I mean, this calculation could take a couple of milliseconds, why not await it? It’s safe, it’s syntactically-cheap… I mean it’s practically free. I’d be a fool *NOT* to await it!

The savvy reader has already noticed that I keep harping on “*syntactically-*cheap” and probably already knows what’s coming next…

Threading in shiro is *not* cheap from a performance perspective. Every time you await something you’re making a copy of all your variables and functions, allocating a bunch of memory, spinning up a second interpreter, temporarily locking your main interpreter’s symbol table and spinning up an OS thread. Then after whatever that thread does is finished you’re re-locking the main interpreter’s symbol table, moving the result over, cleaning up a bunch of memory, tearing down an interpreter instance and cleaning up a thread.

Gross. So you should never use await then because it’s slow and awful, right?

Well, no. See, if you’re actually going to make use of the time that thread is setting itself up and running and cleaning up after itself for other things in your program, then the performance hit of await is actually mitigated almost completely -- your main program is still doing things while the other list is spinning up to evaluate itself. If, on the other hand, you’re just going to like do three milliseconds of work and then sit there waiting for the result of your await… it’s probably not a very good await.

On top of that, there’s the awaith keyword, which does a ‘hermetic await’, where the symbol table is not cloned. Your thread will run with a completely fresh interpreter. This cuts back significantly on the start-up time of the new interpreter, especially if you have a large and unruly symbol table. This still doesn’t mean you can go nuts with awaithing everything, but it helps a lot.

Some use cases for await are obvious… you’re going to use Nimue and you still want to handle user input in your application; you’re calling web-services and waiting for their responses and watch to batch them rather than call them serially; you have a massive batch of files to process and each one is discrete and you want to do 20 of them at a time instead of one.

But the fact that you *can* await anything in shiro can sometimes help you solve some not-so-obvious performance problems. If you have a massive list processing operation or are loading a large file and you don’t need it *right now* and there’s other stuff you can do… a quick await or two can make your program perform like lightning instead of like molasses.

### Implementers and Duck-Typing: Object-Oriented Shiro

Shiro is an object-oriented programming language (cue South Park *rabblerabblerabble*). No, seriously. Polymorphism, Encapsulation, Inheritance and Abstraction are all built right in the language, and those simple inline-objects we’ve been dealing with can become beefy, OO-compliant objects if you want them to.

The reason that few of the examples we’ve seen so far really look object-oriented is because I find I rarely need the full suite of OO principles in Shiro, and the language allows you to mix-and-match as you need them. Mostly I find Polymorphic Inheritance via Implementers and Mixins to be just the right amount of object-orientation for my needs. However, if I want to hide things within other things and build out abstractions… Shiro can do that. It’s just not the way I usually write it, unless I’m writing examples to show off how you can do it.

We’ll start with the easy ones. Shiro is super polymorphic, that’s the whole point of duck-typing. Implementers describe pieces of functionality, but Shiro absolutely never cares *how* you implemented that functionality or even whether or not you mixed-in the implementer or not, it only cares if the thing it’s expecting to be there is there. You couldn’t ask for more polymorphism than that.

Mixins are, essentially, inherited. You can “inherit” multiple implementers to simulate multiple inheritance. If you design your implementers correctly you can have quite complex and functional inheritance chains in your Shiro apps, so there’s inheritance taken care of.

Encapsulation is another pretty low-hanging fruit. You can create an Implementer to represent the outer set of methods and properties you want people interacting with and treat the innards as a complete black-box. By virtue of being both contract/interface and implementation this is quite easy to do and extensible -- you can easily have multiple implementations work adjacent to each other, or a single implementation which also describes the contract.

Abstraction is where things start to get a little rough for Shiro. Shiro doesn’t like secrets; it doesn’t like universal rules; it doesn’t like telling you ‘no’ when it could just as easily tell you ‘yes’, and when you start trying to hide things in the guts of your objects so you can provide only a simple, abstracted set of methods and properties Shiro tends to fight you. Shiro doesn’t want to tell you “hey, that property isn’t there” when it plainly *is* there, just because the guy who designed the object didn’t want you to see it or know it was there.

So I encourage you to stop there, at those three. If you want to signal that a set of methods and properties are not for general public use, I suggest putting them in an \_ property container, like this:

(sod o {

\_: {

private: 1,

private2: 2

},

name: 'dan',

age: 36})

In this fashion you signal your intentions (please don’t touch this stuff unless you know what you’re doing), isolate your “interface” properties and methods from the guts, and still let smart people who know what they’re doing fully work with your objects.

But I know that most developers are a bit more… uh… let’s just say ‘retentive’ about things like that and I’ve already got a hard sell on my hands with this language, so if you just absolutely **must** have your private, secret things that no one can touch, you can by using the enclose keyword. I consider this unidiomatic Shiro, but then my opinion on things like that should be worth very little to you, especially because I acknowledge this one come down much more to personal preference than practical concern.

But just in case you skimmed the keywords chapter (the only other place I use the enclose keyword), here’s how you’d do it:

(sod o (enclose {

private: 1,

private2: 2

}

{

name: 'dan',

age: 36}))

Now none of your friends can get at o.private or o.private2 in any of the normal ways. This being Shiro if I really, really want to I can inject my own object-lambda onto o and access them that way, but at least you did your best to keep them away from me.

### Naming Conventions: My Suggested Approach

I and A/An for implementers (I for plugins/interfaces, A for classes)

kebab-case

predicate naming conventions

“namespaces” using objects. Global namespace containers (perhaps write a shiro library for this?)

# 

# Integrating with Shiro

Prepare yourself for much uglier code samples in this section -- we’ve moved outside the wonderful world of syntax-light Shiro and into the stodgy, static world of C# for this chapter.

We will cover how to integrate with Shiro from a .NET perspective -- that is how to use the interpreter yourself, write your own libraries or host the interpreter in a custom program with built-in autofunctions and autovars. If you only want to write Shiro you can safely skip this section.

## I Lied, Everything is Not a List, Everything is a Token

Well, maybe I didn’t *lie* per se… it’s more like while everything in the Shiro language is a list… everything in the Shiro interpreter is a Token. Tokens can hold literally any Shiro primitive: a string, a number, a lambda, a list, a list of lists of lists, all of your code, etc. etc. In fact the first step to evaluating Shiro is to turn your code into a Token, which is then evaluated.

If you’re going to be writing native Shiro libraries (that is, libraries written in .NET which are used by Shiro), you’re going to be dealing with Tokens, so you’ve got to understand them. And they’re kind of complicated, because as I said they represent literally everything in Shiro.

Let’s take a look at a simple autofunction (a .NET native function which can be called in Shiro), in this case the sqrt function from the math library.

var t = toke.Children[0];

if (!t.IsNumeric)

Interpreter.Error("sqrt function requires a numeric value, not " +

t.ToString());

if (t.Toke is long)

return new Token(Math.Sqrt((long)t.Toke).ToString());

if (t.Toke is decimal)

return new Token(Math.Sqrt((double)((decimal)t.Toke)).ToString());

Interpreter.Error("Internal error in sqrt -- there must be a new numeric type

that I didn't handle.");

return Token.Nil;

toke in this code snippet is the token passed in to the auto function representing the parameter list passed to the function. The Children property of a Token is only set if the token is a list (you can check this with toke.IsParent), and is a list of Tokens which represents the items in the list represented by the token. So the first thing we do is get the first child in the parameter list.

Token has a bunch of .Is\_\_\_\_ properties on it to help you figure out what a token is, like IsNumeric (used here) and IsParent (could be used here if we didn’t know for sure that toke is a list).

## Writing Shiro Libraries

asdasds

## Hosting the Interpreter

asdasds

# Putting it Together

## Writing a MUD in Shiro

asdasds

## Writing a Persistent REST Service in Shiro

asdasds

## Writing a Large, Batch-Processing Application in Shiro

asdasds