How to Shiro

Complete Guide

Contents

[**Getting Started**](#_gjdgxs) **4**

[An Introduction to the Shiro Ecosystem](#_30j0zll) 4

[Why Learn Shiro?](#_1fob9te) 6

[What is this LISP Syntax I Keep Babbling About?](#_3znysh7) 7

[Let’s Play](#_2et92p0) 9

[Glossary](#_d8l4ef49i3ls) 11

[**Shiro From the Ground Up**](#_3dy6vkm) **15**

[Math, Comparison and Variables](#_1t3h5sf) 15

[Control Flow and Functions](#_4d34og8) 16

[Fun with Lists](#_2s8eyo1) 19

[Lambdas and Tigers and Bears](#_lrre6arz40t3) 20

[Objects, Implementers and Prototypes (or: The Art of Monkey-Patching)](#_3rdcrjn) 21

[Interacting with Nimue (the TCP/HTTP/Telnet server)](#_26in1rg) 25

[**Learn your Dev Environments: shiro and shIDE**](#_z7fmrjkswyj9) **28**

[shiro](#_6u41jl3p24px) 28

[shIDE](#_koxcvo6ilwtr) 28

[VS Code / Other](#_3aitpo4xyok8) 28

[**All the Keywords (ugh…)**](#_pda47xhhpwr1) **29**

[Normal](#_3c2ws7wcgip8) 29

[Contextual - Telnet / TCP](#_9v4lpzd0md5f) 29

[Contextual - HTTP](#_csl232f6ka77) 29

[**The Shiro Standard Library (double ugh!)**](#_3omi8yyl7lt7) **30**

[math](#_j43cgmpj4wgb) 30

[**Idiomatic Shiro**](#_lmeo1x1b9ooa) **31**

[With Great Power Comes Great Responsibility...](#_9ned0fxzbvoi) 31

[How Are You Going to Write Your Shiro?](#_3e94i2cb889f) 31

[What Goes Where - Shiro Projects](#_m7xa4qxtvku7) 31

[**Integrating with Shiro**](#_4pzh1qg4mcb4) **32**

[Writing Shiro Libraries](#_29nogvxhfg6c) 32

[Hosting the Interpreter](#_htmbijsvqj9f) 32

[**Putting it Together**](#_k7cofi7x36pt) **33**

[Writing a MUD in Shiro](#_kvfmie1lcakb) 33

[Writing a Persistent REST Service in Shiro](#_c0dnt7uyoq4z) 33

[Writing a Large, Batch-Processing Application in Shiro](#_g1jin96jrbq) 33

# Getting Started

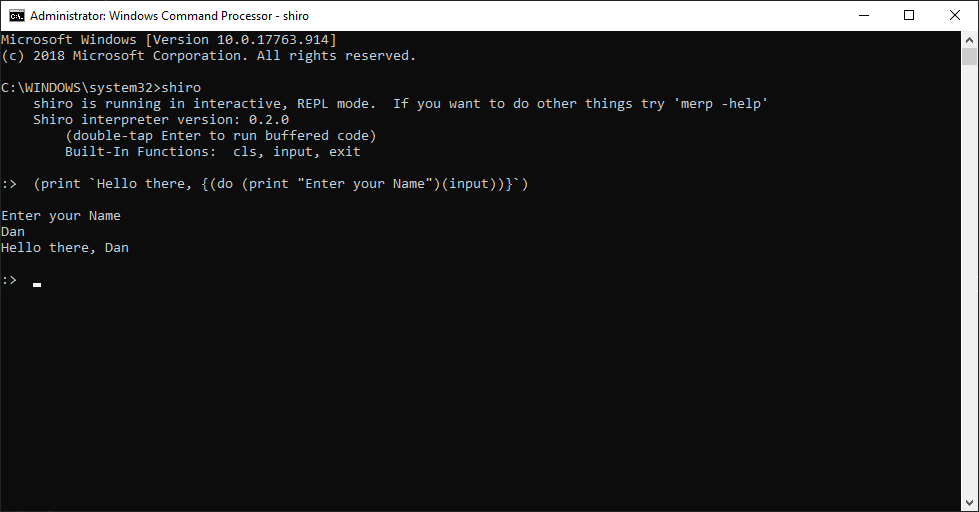
## An Introduction to the Shiro Ecosystem

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.

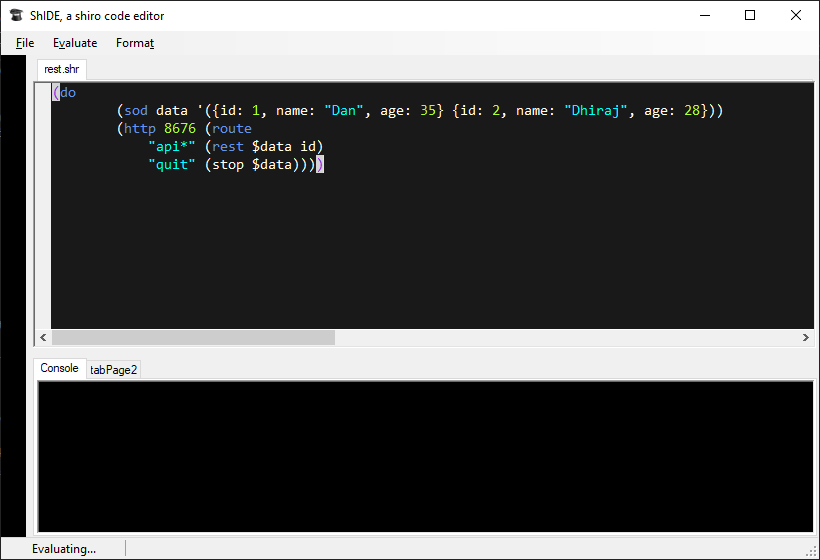


Running ‘shiro help’ will show you a list of commands you can use with it if you don’t just want to run the REPL. Some example commands:

* 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 part 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 know 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…

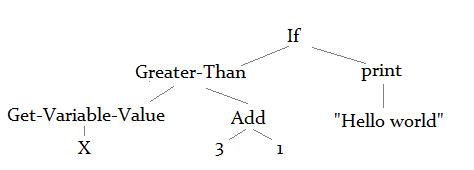
## What is this LISP Syntax I Keep Babbling About?

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 recursive descent algorithms 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!')

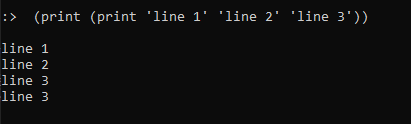
Which if you type and evaluate you'll see has the same result. The parentheses tell Shiro that we're starting a list, and then everything in it is parsed into a separate element. It’s just a simple, text-based representation of that tree-structure we talked about in the previous section.

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.

**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.

**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. 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.

**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*.

**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)

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.

## 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 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 (impl? $o1 IPrintMyself)) ; True

(print (impl? $o2 IPrintMyself))) ; 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.

## 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 multi-threaded 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 $input))  
    (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.

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

asdasds

## Contextual - Telnet / TCP

asdasds

## Contextual - HTTP

asdasds

# The Shiro Standard Library (double ugh!)

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

## math

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?

OO vs FP vs procedural vs. insanity

### What Goes Where - Shiro Projects

asdasd

# 

# Integrating with Shiro

## 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