Shiro Language Quick Start Guide

Version 0.2

Dan Larsen © 2010-2012

Table of Contents

Introduction

Shiro's Purpose

Shiro is an interpreted programming language designed with easy integration with Microsoft .NET application and libraries. In addition, Shiro is a web-server platform designed with an event-loop architecture not unlike node.js (it uses Kayak to provide this functionality under the hood). It is perhaps the only programming environment where it is possible to host a simple web service with a single line of code, and provides many higher-order facilities to make developing such applications easy. It is also many times easier to get initially set up and running that node.js and similar solutions.

This isn't to say there are no applications for Shiro outside of a web project, merely that Shiro's main purpose is rapid deployment, development and prototyping of such. It works just as well as a stand-alone language for any number of tasks, applications and services. The language grew out of a tool which was created for producing "mock web services" as quickly as possible, and is currently robust enough for any number of additional purposes. Despite being an interpreted language (which does not use the DLR), Shiro programs can be compiled into .NET executables with ease, or run in various interpretive environments.

A Programmer's Introduction to Shiro

Some things to note for experienced programmers learning Shiro. The first couple of sections in here are basically just explaining the syntax and special operators and constructs available in Shiro. If you just want to start doing neat stuff and trust yourself to learn from more complex code samples without excessive explanation, you can check out the other tutorials on our website (http://www.shirodev.com). If you're looking for library information, that's at the end.

Anyway, here are some basic, high level things you'll want to know:

•Shiro is fairly strongly-typed, it uses implicit typing and treats all Objects as being assignable to each other.
•All variables are value-types in Shiro: when you assign a variable to a variable or pass it into a function you're creating a copy, and changes will not propagate back to the original. The only exceptions to this rule are the 'this' object and the iterative element of for loops.

•Classes defined inside of functions occur at global scope. Functions defined with the root def (as opposed to anonymous functions) also occur at global scope. Anonymous functions occur at variable scope.
•Shiro does not provide a numbered for loop. Shiro's for loop is an iterative loop for items in a collection. Use a while loop for indexed looping.
•All objects in Shiro can be treated as tupled lists, must like in JavaScript.
•Shiro provides a Read-Eval-Print-Loop (REPL) interface.
•Shiro uses prototypal, multiple inheritance, but provides constraints to allow for limited classical inheritance patterns as well. A new object can be instantiated either from a class or another object, or any combination thereof.
•There are code-injection operators in Shiro that allow for runtime modification of active source code. They can be subtle if you're not expecting them, so you might want to check out the subsections entitled "Everything is a Value (or, Syntax-Injection Hackery)" or "The Parameterization Operator, redux".
•Shiro does NOT allow single-line expressions where it expects a block. The braces must always be present. Let's face it, this "feature" causes way more problems than it solves.

```
#this causes an error
if true
	print 1

#this doesn't
if true {
	print 1
}
```

The Many Faces of Hello World

The first code sample is just about any programming language reference guide is hello world, and so it is with Shiro. It looks just like this:

```
print 'Hello World'
```

Depending on what environment you are running in (Shiro-chan the IDE or shcl the Console), this will print Hello World to the output area. Fancy. Now let's try it in a web browser,

```
use http
httpStart(8081, def() {
	return 'Hello World'
})
```

Run that, then point your browser of choice to http://localhost:8081/. You'll see (no surprise) Hello World. One important thing to note about Shiro's web model is that the server remains running and servicing requests until httpStop() is called, and since we never call httpStop() in this example, the server runs forever. You'll have to kill it manually. If we wanted to be able to do this programmatically, we could try something like this,

```
use http
httpStart(8081, def(uri) {
	if(uri == '/end') { httpStop() }
	else { return 'Hello World' }
})
```

You can still see Hello World at the url we used before, but if you navigate to http://localhost:8081/end the server will terminate. So, that's way cooler than your typical hello world, but let's take it even a step farther. Don't worry if this example doesn't make much sense yet, but if you're coming to Shiro with a background in programming, you can probably see how powerful this is right away. Let's call it, Hello Json.

```
use http
httpStart(8081, def() {
	obj = { Name: 'Hello Json', SomeOtherValue: 23}
	obj = merge(obj, IJsonable)
	return obj.toJson()
})
```

If you take a look at this one in your browser while it's running, you'll see that it returns obj in JSON representation, already perfect to be consumed by any client-side web code using AJAX. There are a couple of advanced concepts in this snippet, like the idea of merging an implementor onto an existing object, but we'll cover that later. Suffice it to say, Shiro is powerful, and writing code in it is FAST.

The Shiro Sandbox

While Shiro is built on .NET and integrates readily with .NET assemblies, it should be noted that its development environment is a custom sandbox. You will not consume .NET APIs or namespaces directly, but rather deal with a simplified and extensible library of native code. This decision was made due to the sheer size and complexity of the .NET library. Shiro is meant to be simple, fast to develop, easy to learn and somewhat higher-level than typical .NET code. Especially with it's complex type system, providing any kind of interoperability with the entirety of the .NET library would introduce too many semantic constraints and syntactic problems to really consider.

The Shiro Standard Library provides much of the functionality you might require from .NET, and grows at a consistent pace in line with user feedback. It is also easy to add your own assemblies to integrate with existing applications, or just provide some additional functionality that the core library is lacking. There is a bit of class decoration with attributes that is necessary to make things callable from Shiro, but it takes only a few minutes to make any .NET assembly export library methods that the Shiro runtime can consume.

There are several "tiers" of the Shiro Standard Library, the bedrock of which are two separate libraries, Core and Std. Both of these libraries are referenced automatically by Shiro, so you gain access to their functionality immediately in any Shiro program or script. Core is a set of native routines which almost all significant programs will use at least pieces of, and Std is a library written in Shiro itself which provides some incredibly useful constants, implementors and methods. The IJsonable implementor shown in the Hello Json example above is defined in the Std library.

The second "tier" consists of several additional libraries come bundled with the Shiro Interpreter (and of course additional libraries can be loaded by DLL path trivially). Some are native code and others written in Shiro (some, like Thread and Http are a combination of both). These libraries are:

Assert: A typical assertion library for writing tests
File: File and File System IO
Http: The node.js style, event-driven http server module
Math: Just what it says on the tin
SQL: Methods for dealing with SQL databases
String: ...
Thread: A simple and powerful threading system designed to make writing thread-safe code as easy as possible.
Xml: Give you one guess...

Finally there is a third tier of user-defined libraries, which are those you roll yourself. In addition, all known Shiro interpreter providers (ie: shiro-chan, shcl and the compiler) provide a library known as Console which exposes some basic IO mechanisms for user interaction. While the existence of the Console library is standard in all of my personal implementations of Shiro, it is quite possible for someone to write one without a Console library, so don't assume it's a standard across all platforms.

Fundamentals of Shiro Syntax

While Shiro shares scope notation with many languages like C, C#, Java and JavaScript, Shiro minimizes unnecessary language clutter where possible and uses many constructs differently, so don't be deceived by the curly-braces everywhere. Comments in Shiro are delimited with the pound (#) character, and code starts executing right at the top – there is no need for a main function or an object at the root level. Semi-colons are strictly prohibited, and line-formatting is essentially whatever you prefer – Shiro has no rules about it. You can do multi-line comments using #> <# (with everything between them being ignored).

Flexibility was paramount when designing Shiro's syntax; there is no 'one-true-way' to write in the language, and where possible it'll work around your preferences and comfort levels. You can write Shiro in REPL, without structure at all, procedurally like a C program, or in two rather distinct flavors of OO (classical and prototypal). A great deal of time and care has been taken to create a cohesive language out of such flexibility rather than just a collection of strange semantics. While brevity is valued, clarity is even more important.

Types and Variables

There are no formal type declarations in Shiro; a variable comes into existence when a value is assigned to it; despite this, Shiro is a strongly-typed language that enforces type safety. If you assign a variable a number then later try to assign that variable a string, Shiro will throw an error to let you know you messed up. Names in Shiro are case sensitive, meaning F and f and two different variables.

The following code demonstrates this, with the comments serving as useful annotation.

```
X = 123                        # X is a number
X = 'Hello world'              # error:  type mismatch
convert X = 'hello world'      # this changes X's type to string
X = 123                        # error, type mismatch
print X is String              # true
print X is Number              # false
```

This examples introduces you to String and Number, two of Shiro's native types. It should be noted that, unlike in most object-oriented languages, these native types are not themselves objects (with the exception of the Object type, of course). The remaining native types are:

Bool: Can only be either 'true' or 'false'. Any other type can be treated as a bool (for example, used in an if statement), with 0 or null generally representing false and any other value representing true.

Number: A number (either integer or decimal – the internal type varies as necessary).

String: Strings can have embedded newlines. In addition, strings may be marked with either single or double quotes, although this must be consistent on a per-string basis. This allows you to embed quotes of the other type inside the string. Both of the below are valid:

```
s = "This is the first line,
       and this is the second line of the string"
s = 'Dan said, "Hi there!"'
```

Function: You can assign any function to a variable, even a member of an object, a library function or an anonymous function (one you create just to assign to a variable).

List: A more dynamic implementation of the array available in most languages, Shiro lists are very dynamic and can contain any number of other variables. They grow and shrink dynamically as members are added to or removed from them, and are not restricted to a single type. You can declare lists in line, like so:

```
L = {1,2,3, 'value', true}
```

Object: Objects are the root of object-orientation, an instance of a class which contains both properties (variables) and functions (methods). Explained in more detail later on. Like lists, objects can be declared in line, like so:

```
O = { name: 'Dan', age: 28 }
print O.name              # Dan
```

Class: A Class in Shiro is the template for an object, defining the default properties and methods the object will contain. It should be noted that objects can be created without classes if one chooses to use prototypal OO instead.

Control Flow

Shiro supports a standard set of control flow operations, including loops, conditionals, structured exception handling and function calls. It is somewhat outside the scope of this reference to provide an in-depth tutorial on basic

programming concepts, but differences from expectations will be touched on as we briefly consider each control-flow keyword. The if keyword works in familiar fashion, although it is not required to put the condition in parenthesis as in most languages derived from C. Although you can if you want to I suppose. Else and else ifs are both syntactically valid,

```
if true {
        print 'this will print'
} else {
        print 'this will not print'
}
```

There are two looping constructs in Shiro, while loops which loop while a condition is true, and for loops that loop through every object in a List (in many languages, Shiro's for loop is called a 'for each' loop). The code example below shows two different ways to print the numbers 1 through 10.

```
i=1
L = { }     #we'll build this inside the while loop
while i<=10 {
        print i
        L += i
        i += 1
}
for item in L { print item }
```

There are quite a few new concepts in this example, so let's take a closer look at it. First, we're creating some variables at the top; specifically, a list (called L) and a number called i. You'll note that putting things into a list is as simple as adding it with the standard addition operator. You'll also notice that compound-assignment operators like += are available.

There is no equivalent to C/Java's for loop in Shiro, as it is quite easy to implement a counting-loop, as shown, and the most common application thereof is iterating over items in a collection, which is handled by the language's for loop. In addition, the for loop has a handy little additional feature: you can constrain the loop using a where clause, which will only execute the loop if the constraint is true. A simple example which prints everything in a list with a value greater than 3,

```
L = {0,1,2,3,4,5,6,7,8,9}
for item in L where item > 3 { print item }
```

Most programming languages have some form of switch or select command, allowing one to select over a range of choices. Shiro is no exception, using the select keyword, with options inside for the different values. The example below

shows this, as well as defining a simple function

```
def GetName(i) {
   select i {
      option 1 { return "one"}
      option 2 { return "two"}
      option 3 { return "three" }
      option else { return "more than 3 (or less than 1 I guess)" }
   }
}
print GetName(1)
print GetName(6)
```

You'll notice we're actually declaring a function here called GetName. Don't worry, we'll talk more about functions momentarily.

You can break out of a loop, menu or conditional using the break keyword. If you want to break out of multiple levels of construct, you can use 'break twice' or 'break thrice'. The omission of break for more than three levels of nesting is both to encourage better coding practices and because the author is not certain what the equivalent to thrice is for four of more elements. If you ever find yourself breaking thrice, there's almost certainly a better way to code what you're writing.

Shiro's try-catch-finally pattern is fairly standard, with the addition of an optional 'else' block, which will only be fired if no exception occurs. The various blocks in this construct are:

try: Code in this block will be executed first. Based on the behavior of this block (specifically, whether or not it throws an exception), additional blocks will or will not be executed. Required.

catch: Should the try block throw an exception, the code in the catch block will be executed. The catch block is required.

else: An optional block which executes if the code in the try block did not throw an exception.

finally: Regardless of whether or not there was an exception, this optional block will execute after all other blocks are run.

While Shiro will throw exceptions itself when invalid operations are encountered, it is also quite possible to throw an exception yourself using the throw keyword. Here's an example covering all these instances:

```
try {
      throw "an exception"
```

```
} catch error {
	print "The exception was: " + error
} else {
	print "This will not print."
} finally {
	print "This will always print."
}
```

Note that catch must always be followed by a variable name, which will receive the thrown exception. Even if you don't plan on using this value in the block, you will need to include a name for the error to be placed in.

There are a couple of keywords for including libraries or running external code. You can import a Library using the use keyword.

```
use Std			# it is never necessary to do this explicitly for Std
use Console
use MyLib in "C:\MyAssembly.dll"	# Explicit load
```

If you want to explicitly run a Shiro script (and don't want to load it yourself and use eval or the Execution operator) you can do so using the run keyword.

```
run "C:\script.ml"
```

Lists

Lists are a major part of Shiro's flexibility. As we progress farther one and get into complex object-oriented functionality, it is key to remember that the most complex object can be treated as a list if it is more convenient to do so. This idea was shamelessly cribbed from Javascript and built up from there into something quite unique. This flexibility is accomplished (without making the syntax itself too confusing) by providing several different ways to access and manipulate lists.

In their simplest forms, lists look a lot like arrays in other languages:

```
L = {1,2,3,4}
print L[0]		#Will print 1
print len(L)		#4

print L[-1]		#4
print L[0 to 2]		#123
print L[2 to 0]		#321
print -L		#yes, this works too, prints 4321
```

In addition, you can add and subtract things to lists with the normal operators. This includes adding and subtracting whole lists from each other. You will find this works quite intuitively once you get used to it, such as,

```
L = {1,2,3,4}
L = L + 5          #L is now = {1,2,3,4,5}
L = L - 2          #L is now = {1,3,4,5}
L = {6,7} + L      #L is now = {6,7,1,3,4,5}
```

Functions

The Shiro def keyword serves double-duty for defining both static and anonymous functions. You can use it at the root level to define a static function, or you can use it in an expression to create an anonymous function. Here's two different ways to write a simple function that returns 1:

```
def F1() { return 1 }
F2 = def() { return 1 }
```

At a glance these look pretty similar, and both functions work the same exact way. You can call them interchangeably, and both will return 1. The difference is F2 is a variable, you can assign it a different function later and do many other interesting things with it. You don't even really have to give it a name. Many functions in Shiro's libraries take functions as arguments, like forAll. Here's the forAll function, from the Std library:

```
def forAll(L is List, F is Function) {
        for item in L {
                F(item)
        }
        return L
}
```

By the way, as that example demonstrates, you can use the is operator to constraint arguments. More on that later, but the basic gist of it should be fairly obvious from this example. Let's see forAll in action, notice how we're passing a whole new function in for F,

```
L = {1,2,3,4}
forAll(L, def(i) {print i })
```

Pretty neat, although most languages can do this kind of stuff now, even the stodgy ones. Of course Shiro lets you do all the standard stuff like recursion and whatnot. Here's a Fibonacci example,

```
use Console
cls()

def Fib(x is Number) {
	if x <= 1 { return 1 }
	return Fib(x-1) + Fib(x-2)
}

L = {1,2,3,4,5,6,7,8,9,10}
forAll(L, def(i) { print Fib(i) })

#or we can put the results into a list:
result = {}
forAll(L, def(i is Number) { result += Fib(i) })
print result[0 to 9]
```

Although you don't have to, you can specify a type for arguments using 'is', as shown in the forAll function above. If you do so, Shiro will throw an exception if someone passes in a badly-typed value. You can specify a basic type (Number, String, Function, Bool, List, Object) or a class name (more on this in the object-orientation chapter, don't worry). The choice of how 'loose' to play with your argument typing is a personal one, for a simple one-off application you can generally avoid type constraints.

Anonymous functions can also be parametrized using the ParamOp (@). This allows you to build functions from templates, and Shiro parameters are not unlike SQL parameters in use, check it out,

```
def CreateAFunctionThatReturns(retVal){
	return def() { return @retVal }
}
F1 = CreateAFunctionThatReturns(1)
F2 = CreateAFunctionThatReturns("Hello world")
print F1()	#1
print F2()	#Hello world
```

Once created a parametrized function performs just as fast as any other function call, so don't be shy about making use of the ParamOp to facilitate meta-programming. Keep in mind though that you can only use the ParamOp inside of an anonymous function, anywhere else and it just won't work; after all, other blocks are static and can't change like anonymous functions can. It's important to keep in mind as you learn more about Shiro that anonymous functions are extremely dynamic. This is just one, relatively mild example.

Functions can take a variable number of arguments. To do so, use the following syntax:

```
def HowManyArgs(...) {
	print len(args)
}

HowManyArgs(1,2,"Dan was here")
```

The … must be the last item in the argument list, and any supplied arguments will be accessibly inside the function via the args list, as shown above. One final syntactic shortcut, you can actually do some stuff after the return without effecting the return value using the optional 'then' clause, like so:

```
return retVal then {
	retVal.cleanUp()  # won't effect the actual returned value.
}
```

Objects

In order to really understand the power of Shiro's object orientation, it's necessary to understand that objects can be treated in many different ways. Despite all the constraints and options we'll be introducing shortly, it's important to remember that at their core, Shiro objects are just lists; lists indexed by name instead of number.

There are a bunch of ways to create objects, the simplest is just to assign one to a variable, like so,

```
Dan = {
	name: "Dan",
	age: 25,
	sex: "Male",
	printName: def() {
		print name
	}
}

#all of these do the same thing:
print Dan.name    #access property
Dan.printName()   #call method
print Dan["name"] #access as a tuple
print Dan[0]      #access as a list
```

Remember, even though they look like objects, the underlying values remain lists. You loop through an object just like you can a list (although depending on the object this can become rather nonsensical). It does make it very easy to display

the contents of an object though. Using the tuple() function in Std, which returns a list containing all the names used in the tuple, you can do things like:

```
for tn in tuple(Dan) where !(Dan[tn] is Function) {
	print tn + " = " + Dan[tn]
}
```

So far these objects are kind of lame to, say a C# programmer. You declare them inline and they're basically static classes; we haven't instanced an object yet, but let's rectify that. Instead of making a Dan object representing one guy, let's apply some design here and make an abstraction, Person. Please note, we're still not dealing with actual classes yet... Person is an object in the following code example just like Dan was above.

```
Person = {
	name: ""
	,age: 0
	,sex: "Neuter"
 }

Dan = new Person
Dan.name = 'Dan'
Dan.age = 27

print Dan
```

Wait, you might object now, Person is obviously a class in the example above... we even create one with the new keyword, just like in your static OO language of choice. The difference is that Person can be mangled like any Shiro object... you can add stuff to it, change it's properties around, merge it with other objects, whatever. These changes will not effect objects created from Person, since the new keyword in this instance is only creating a copy for you to operate on.

If the class keyword prefixes an object declaration, and the resulting object cannot be used directly. Its properties and methods are all there (the class is, like most everything in Shiro, just a tupled list), but you simply can't get at them. Classes have to be instanced, either using inheritance or using the new keyword, before they can be used. Behold,

```
class Person = {
   name: ""
   ,age: 0
   ,Person: def(n,a) {
      name = n  age = a
   }
}
```

```
class PrintablePerson := Person, IStringable {
   printIt: def(){
     print this.toString()
   }
   ,PrintablePerson: def(n,a) {
     this.Person(n,a)
   }
}

dan = new Person('Dan', 27)
bob = new PrintablePerson('Bob', 40)

bob.printIt()
print dan.name

print dan is Person   #true
print bob is Person   #true
print bob is PrintablePerson   #true
print dan is PrintablePerson   #false
#Person.name = "Dan"    # this throws an exception
```

There are a bunch of new things in this code sample, so let's go over some of them. The first is the addition of a constructor to the two classes here. By adding an anonymous function with the same name as the class or object, you can subsequently use it with the new keyword, as shown. Note that even if a constructor is defined, you can still use new without any arguments to create an unconstructed instance of the class.

Next up, we have inheritance going on with PrintablePerson. PrintablePerson combines the Person class with the IStringable implementor, which makes an object able to convert itself to a string in a standard way. You can inherit as many objects, classes and implementors as you want. You can even merge them in after the fact in many cases (such as in our Hello Json example in the introduction).

A note on inheritance... while you can multiply-inherit, only the first class in the list goes into the inheritance chain. That is, while the inheriting class will be of the same type as the first base class (and all of its first base classes), it will not be of the same type as any other bases. The 'is' operator checks for base class while =? checks for equivalence. A class will always be equivalent to every class it inherits from. This might clarify a bit:

```
Person = {
  name: "Dan",
  age: 23
}
```

```
Dan := Person, IStringable {
  gender: "Male"
}
print Dan is Person     #true
print Dan is IStringable   #false
```

To get around this limitation (and for a myriad of other purposes) there is the equivalence operator, or the does-it-quack operator, if you will, =?. This checks for object equivalence rather than direct inheritance. Continuing the code snippet above,

```
print Dan =? IStringable   #true
```

Also note that the equivalence operator can be used as a function argument constraint, just like is.

```
def printIt(o =? IStringable) { print o.toString() }
printIt(Dan)
```

Note that the example above would not work using an 'is' constraint as Dan's base class here is 'Person', not IStringable. The combination of these two argument-type constraints allows you to make functions which take arguments as general or specific as you'd like.

Everything is a Value (or, Syntax-Injection Hackery)

Everything in Shiro is a value type. When you pass arguments, you're passing copies, not originals. When you assign a variable to another, again, you're copying it. There are no reference types or pointers, although as with most things in the language, there is a way around this problem if you absolutely must.

There are two unique operators in Shiro, the name operator ($), which is just a shorthand-way of making a single-word string out of a variable name, and the execution operator (~), which inserts the value of a variable directly into the token-stream at the time of interpretation (and yes, that is as powerful and potentially irksome as you imagine it might be).

The most common application of these two options is to use them to basically pass references (by passing the actual name of the variable and not a copy of it), like so:

```
def Increment(v) {
     ~v = ~v + 1
}
```

```
X = 1
Y = 10

Increment(X$)      #or Increment('X')
print X       #2
Increment('Y')
print Y            #11
```

It can be a little hard to wrap your mind around, but basically at the time Increment is called, it inserts whatever string is passed in as its argument where ~v is.

This can be used for far more than just variable names (although again, the clarity and safety of the resulting code can be somewhat weak). Here's an eval method:

```
def execute(code) {
	~code
}

execute("print 'hello world'")
```

The execution operator can appear at any point in the token stream, and supersedes all other operations – as long as the result of inserting the string into place is syntactically-valid Shiro, it runs just fine. This can lead to some interesting code, like the function below that returns the named property of an object (a pretty useless function considering you could just square-bracket it), but still,

```
def getp(o, propName is String){
	return o~('.' + propName)
}

thing = {name: "bob"}
print getp(thing, "name")
```

If you never, ever use this pattern, you certainly have my blessing.

However if you use it in cool and strange ways, E-mail me the code so I can enjoy it too :).

The Parameterization Operator, redux

I just wanted to touch once more on the param operator (@), which we covered

in the section on Functions above. Specifically, I'd like to show you how it's different from the execution operator (~) that we just learned about.

The two are quite similar, except the parameterization operator is evaluated at the time the function or block is defined and stays the same for the rest of it's lifecycle, while we've seen that the execution operator actually drops its payload every time. Also, the parameterization operator can only be used inside an anonymous function. This brain-bending bit of code should highlight the differences:

```
X = 123
F1 = def() { print ~X }
F2 = def() { print @X }

X = 999

F1()   # will print 999, the *current* value of X
F2()   # will print 123, the value of X at the time F2 was defined
```

Operator Chart

The following table contains all combinations of operator/operand type, and what happens with the operation. The obvious (like Number + Number) have been omitted:

Operator	Type 1	Type 2	Result	Example
+	String	Number	String	"Hello" + 23 = "Hello23"
+	List	Num/Str	List	{1,2} + 43 + "Bob" = {1,2,43,"Bob"}
+	List	List	List	{1,2} + {3,4} = {1,2,3,4}
-	String	Num/Str	String	"Hello world" - "world" = "Hello "
-	Number	String	String	123456 - "34" – "1256"
-	List	List	List	{1,2,3,4} - {1, 3} = {2,4}
-	List	Num/Str	List	{1,2,3,"Bob"} - 2 - "Bob" = {1,3}
*	String	Number	String	"Bob" * 3 = "BobBobBob"
*	List	Number	List	{1,2} * 2 = {1,2,1,2}
/	String	String	List	"break into words" / " " = {"break", "into", "words"}
/	List	List	List	{1,2,3,4} / {2,4,5,6} = {2,4} (common elements)
/	String	List	List	"1,2|3" / {',', '|'} = {1,2,3}

Object Oriented Principles

Class vs Object vs List

I keep harping on this point, and I'll try to stop now, but at the heart, every Shiro object and class is nothing more than a tupled list. The language enforces certain constraints and use patterns based on what exactly the object was, how it was declared and how you're trying to use it. This little snippet shows you how the different types are reflected with the is keyword:

```
o = {name: ""}
class c = { name: ""}
print o is Object & o is List   #true
print c is Class & c is List    #true
print o is Class | c is Object  #false
```

To extrapolate a bit, depending on how you try to access a given object or class determines what Shiro will and won't allow. If you use the dot (.) notation, you're enforcing OO constraints and, for example, can't access protected members (any member whose name is prefixed with an underscore). If you use list notation ([]) however, you can, as this example demonstrates:

```
use Assert
o = {_prot: "Dan"}
assert.hadException("Can't access protecteds using dot notation", def() {
      o._prot = "Bob"
})
o[_prot$] = "Bob"   #this works though
```

Lists don't have protected indexes, but objects do have protected members. Instead of trying to choose one of the other and breaking away from Shiro's "everything is a List" approach, the language simply provides two different paths with different functionality. You can even access Class members using the list notation and modify them (normally a Class cannot be modified except by inject()), although the uses of this are fairly limited, and if you find yourself doing it often you probably don't need to have classes at all.

Inheritance

There are two kinds of inheritance, Classical and Prototypal. Classical is the kind you're probably familiar with coming from C++-based OO languages. The

classical approach to creating an object is to:

•Define the structure of the object, using a class
•instantiate that class to create a new object.

Objects created in this manner have their own copies of all instance attributes, plus a link to the single copy of each of the instance methods. Classes are definition, Objects are implementation and nary shall the two meet.

In prototypal inheritance on the other hand, instead of defining the structure through a class, you simply create an object, which can be used on its own or be reused by new objects. It is called the prototype object because it provides a prototype for what the other objects should look like.

Shiro does both kinds of inheritance, as we've already seen to some extent. Objects in Shiro are very dynamic, they can change, have new objects combined with them, have interfaces injected into them and a half-dozen other such things. Prototypal inheritance is a more natural fit for this kind of environment, but efforts have been made to provide some interpreter support for using classical constraints, and there are many hard-interfaces where a class is more appropriate than a loosey-goosey object.

Let's look at some examples:

```
class c = {
        name: "Dan",
        c: def(n):
                this.name = n
}
c.name = "Bob"    #fails
c[name$] = "Bob"   #works
aC = new c('Bob')

print aC.name     #bob
o = {
        name: "Dan",
        o: def(n):
                this.name = n
}
o.name = "Bob"  #works
anO = new o("Bob")  #also works
```

Both c and o in the above example can be inherited, with mostly the same results. If c is inherited by an object, the object that inherits it is not necessarily a class (unless declared using the class keyword). Shiro also allows for multiple inheritance without issue.

Implementors, Shiro's answer to the Interface

You've seen IStringable and IJsonable in code samples already, and heard me talk about implementors. Implementors are Shiro's answer to interfaces and conceptually serve the same purpose. Interfaces are just contracts, containing no implementation, and Shiro doesn't do pure definition like that... Even class members have values (even if just the default for their type), and the language doesn't have the concept of a signature without something behind it or a null value.

Still, it is often nice to pick up discreet packets of functionality and be able to either get the implementation for free or check in future code to see if an object supports a set of functionality.

IStringable is a pretty simple implementor, it just provides a .toString() method which returns a string representation of the object. Many library functions (like output) are aware of this convention and will use it, so you can write objects that print prettily. Unlike a traditional interface, however, IStringable doesn't force you to implement .toString() on your own, but rather brings its own default implementation (which just prints the properties on an object) which you can override if you want or leave as-is. That's the critical difference, Implementors bring functionality, not just the promise to implement functionality later.

Remember that the is operator only reflects the first class in the inheritance list (and the first object in its inheritance list, ad nauseum), so if you inherit multiple implementors, it is incorrect to check for them using is. Instead use the Equivalence Operator (=?). This example should help to clarify things:

```
B = { name: createA(String) }
C := B, IStringable {
        name: "Dan"
        ,age: 27
}
print C.toString()
C.toString = def() {  return "My name is " + name }

print C.toString()
print C is IStringable     #false, the correct way to check is:
print C =? IStringable     #always true, even if we multiply inherit.
```

Implementors can be either objects or classes. While the Shiro standard library uses classes, this is purely a stylistic choice and not a requirement. If an implementor can be used in a stand-alone way, feel free to make it an object instead. The only thing that makes an implementor an implementor is the I in

front of the name, and that's just a convention I picked up. Here's the entirety of IStringable as defined in Std:

```
class IStringable = {
	toString: def(){
		sConv = ""
		for key in tuple(this) {
			if !(this[key] is Function){
				sConv = sConv + format('%0 = %1%2',
					key, this[key], NewLine)
			}
		}
		return sConv
	}
}
```

Polymorphism

With it's implicit typing and loose default type-safety for objects (that being, all objects are Objects and can be assigned to that type of variable), Shiro is an extremely polymorphic language. Without constraining, objects can be passed around without type checking at all, although this is bad practice it highlights the fact that Shiro is, by default, permissively polymorphic. Any object can serve as any other as long as the expected methods and properties are there; a condition you can easily check for with the equivalence operator.

The use of type constraints lets you write much safer Shiro, but even with type checking the language retains polymorphism. An object matches a type constraint for its full inheritance chain, and if you use the equivalence constraint it can even do normal matching for implementors or just functionality packets. For example:

```
#normal constraint
def printA(o is A) {
	print o
}

#equivalence constraint
def printObj(o =? IStringable) {
	print o.toString()
}

A = { name: ""}
B := A, IStringable { age: 27 }
```

```
#these all work
printA(A)
printA(B)
printObj(B)
```

Reflection

Shiro is not shy about self-reflection, and interrogating the symbol table and programming environment is an important part of writing code in the language. All objects and classes will readily provide a list of their properties and methods and allow them to be mucked with. There is no reflection library, nor any convoluted syntax to use for it. The fact that objects are lists and that the Shiro library provides a bunch of ways to deal with them and scan them down is all you need. Shiro even provides a function, getGlobal() which returns the entire Symbol Table as a single list. Check it out:

```
globalVar = "Hello world"
print getGlobal().globalVar
```

To get a list of named indexes on an object, just call the tuple() function. This can be seen in the IStringable listing above, in the line:

```
for key in tuple(this) {
…
```

Method Injection and Object Merging

Two of Shiro's secret sauces are the ideas of injection and merging. In designing these areas of the language the intent was the preserve the functionality of objects in JavaScript (allowing, for example, properties and methods to be added and changed at runtime) while eliminating many of the common error conditions caused by JavaScript's total lack of sanity checking. Basically, you can't just add things to an object by name:

```
C = {name: "Dan"}
C.age = 25        #fails
C[age$] = 25      #fails
```

Often in languages which allow this construct (doubly-so for case sensitive languages) a typo or a capitalized letter can inadvertently change object definitions and create errors which are very hard to track down. Shiro solves this by forcing you to use a different, explicit mechanism to inject properties and

methods into an object (while still allowing and encouraging this behavior). These methods are the inject and merge functions in the Std library. You can inject a single method or property into an object using inject, as shown below:

```
Person = {
	Name: "dan"
	_age: 27
}

Person = inject(Person, 'doIt', def() {
	print Name
	print _age
})
Person.doIt()
```

Notice how we gain the implicit this and access to protected members of Person inside the doIt method, even though it is technically not defined inside the object. Once injected the new member acts in every possible way as a part of the object it belongs to. Note that injecting into a class or an object from which other objects inherit will not change any existing children/instances, only future ones.

Injecting single methods and members is neat, but the real cool stuff is the ability to merge entire objects. Let's take another look at that Hello Json example from waaaaaaay back when,

```
use http
httpStart(8081, def() {
	obj = { Name: 'Hello Json', SomeOtherValue: 23}
	obj = merge(obj, IJsonable)
	return obj.toJson()
})
```

Notice how obj is just defined, without inheriting anything, inline. We subsequently merge it with IJsonable, and suddenly obj has a toJson() method (and a fromJson for that matter).

By breaking away from traditional, OO design patterns and thinking in terms of implementors and packets of functionality, it is possible to build robust and flexible solutions with very dynamic objects and classes. Code reuse is also a huge factor here, and well-architected Shiro will often provide dozens of useful implementors and code pieces which can be used again in future projects or built into a library.

Or, you know, just stick to writing objects and classes. Or just functions. At the end of the day, you can get from here to there in any number of ways using Shiro.

Standard Library Reference

Shiro's interpreter comes bundled with a number of useful libraries, which collectively make up the Shiro Standard Library. It is possible to bind to native libraries in other assemblies or to load Shiro source files. By convention, all Shiro interpreter hosts should include the Std library (which includes Core as well). There is rarely a need to explicitly 'use Std' yourself, but if you use a 3rd party interpreter host you may have to load it explicitly. Other libraries can be loaded with the use keyword on an as-needed basis.

You should also never have to 'use Core'. If for whatever reason you're in an environment that does not load Std by default, you should 'use Std' to include the functionality in Core. The reason for the division is that Std is a Shiro-language library and Core is a native one, the two are intimately related, and from your perspective the Core library is essentially invisible, its methods exposed when you use Std. This pattern of intermixing Shiro code with native is used quite often throughout the interpreter -- the Shiro code including the native libraries it needs, with the developer 'use'ing the Shiro library and not the native one directly.

Std

Std provides the following useful constants:

Type Constants: Function, Number, String, Boolean, List, Object, Class

Other: NewLine

Std also provides a set of very common Shiro implementors,

IStringable (class IStringable = { toString() }): Objects which inherit IStringable gain a toString() method (which can be overriden) that returns a string representation of the object. Default implementation just lists property values, as shown in the following code sample:

```
C := IStringable { name: "Dan" }
print c.toString()

# or

output(C)
```

IJsonable (class IJSonable = { toJson(), fromJson(json) }: This

implementor, as the name suggests, can render an object to JSON, or load an object from JSON. IJsonable can be used from the root to initiate an object from JSON, or off an object to fill it with JSON.

```
P := IJsonable {name: "Dan", age: 27}
json = P.toJson()
print json
P.name = "will"
print P.toJson()

# once we fromJson on P, name goes back to what it it is the json

print P.fromJson(json).name   # Dan
print P.name                  # Dan
```

You can also initialize a data object from json like this:

```
newP = new IJsonable
newP.fromJson(json)
print newP.name         # Dan
```

The following methods are provided, either by Std or Core, when use-ing Std:

abs (abs(val is Number)): Returns the absolute value of val

allTrue (allTrue(L is List)): Returns true if everything in list L evaluates to true, false otherwise

applyToAll (applyToAll(L is List, F is Function)): Works just like forAll, except that it applies the result of function F to each item in the list. The implementation of this method is:

```
define applyToAll(L is List, F is Function) {
        for item in L {     item = F(item)    }
        return L
}
```

base (base(obj is Object)): Returns the base class name (as a string) of obj

createA (createA(type)): Creates a new variable of the type-name passed in. Useful for initializing class members

eval (eval(code is String)): Evaluate code in a separate instance of the Shiro interpreter (with separate symbols). Vastly less efficient than simply using the execution operator to accomplish the same thing (although you may sometimes need the separate symbols.

```
eval('print "Hello world"')
```

executeApp (executeApp(appPath is String)): Run an application or URL

forAll (forAll(L is List, F is Function) | F = def(item)): Calls F for each item in L

forAllIdx (forAllIdx(L is List, F is Function) | F = def(list, index)): For every item in L, calls F with L and the index of the current item. Like so:

```
use Assert

n = 123
s = "bob"
o = {name: "dan"}
l = {1,2,3}
b = true

f = def() {    print "Hello world"    }
vals = {n, s, o, l, b, f}
types = {Num, String, Object, List, Bool, Function}
notTypes = {String, List, String, Object, Num, Object}

forAllIdx(vals, def(list, i) {
	checkTrue = types[i] checkFalse = notTypes[i]
	assert.isTrue(list[i] is ~checkTrue, 'is check ' + types[i])
	assert.isFalse(list[i] is ~checkFalse, 'is check (false) ' + types[i])
})
```

format (format(fmt, ...)): Format takes string fmt and any additional arguments. It uses %# (starting with 0) as format placeholders and returns the resulting string. Handles IStringable classes in the args list.

```
name = "Dan"
age = 27

print format("My name is %0 and I'm %1 years old", name, age)
```

free (free(varName is String)): Releases the symbol whose name was passed

in varName from the current Symbol Table.

getGlobal (getGlobal()): Returns the global symbol table as a tuple. Used mainly for creating new threads, but could be used for other things (ie: saving application state).

getMatch (getMatch(L is List, comp is Function) | comp = def(item)): Returns a list of everything in L which returned true when passed to comp.

getUid (getUid()): Returns a new GUID in a string.

inject (inject(obj is Object, name is String, val)): Return an object based on obj, with additional field name, of value val.

```
P = {name: "Dan" }
P = inject(P, age$, 27)

print P.age  # 27
```

inList (inList(L is List, val)): Returns true if val is found in L

insertAt (insertAt(list is List, pos is Number, val)): Inserts value val into list at index pos and returns the result.

len (len(val)): Returns the length of a string or list.

merge (merge(primary is Object, secondary is Object)): Combines two objects and returns the resulting object.

ms (ms()): Returns a millisecond tick count that can be used for benchmarking or timing.

output (output(val, ...)): Performs a format and prints the result.

rand (rand(maxVal is Number)): Returns a random number between 0 and maxVal.

sleep (sleep(duration is Number)): Pauses execution for duration milliseconds. Yields control to other threads.

sort (sort(list is List)): Returns a sorted version of list.

tuple (tuple(list is Object)): Returns a list containing the names of every property/index in list. Useful to get property/method names from objects.

```
Dan = { name: "dan", age: 27 }
for tupe in tuple(Dan) {
```

```
        print tupe + " = " + Dan[tupe]
    }
```

type (type(val)): Returns the type name (as defined by the type constants above) of val.

File

The File library, as you might guess from the name, handles file IO in Shiro. It is based around the (somewhat old-school) idea of file handles, that is, you call one of the open methods and it returns a handle, which you pass to subsequent File calls to tell it which file to operate on. It would be quite possible/easy to wrap this library in a Shiro class representing a file to give a more OO approach, but this works for most of my needs pretty easily.

close (close(handle)): Closes the file whose handle is passed in.

eof (eof(handle)): Returns true if the file whose handle is passed in is at End of File.

fileExists (fileExists(name is String)): Checks to see if a file exists in the file system.

openr (openr(name is String)): Opens file name for reading, and returns the handle to the opened file.

openw (openw(name as string, append is Bool)): Opens file name for writing, and returns the handle to the opened file.

read (read(handle)): Reads the entirety of a file

readln (readln(handle)): Reads a line out of a file

write (write(handle, value)): Writes something to a file

writeln (writeln(handle, value)): Writes something to a file and follows it with a NewLine.

String

Following the convention of naming Shiro libraries after what they do, the String library deals with manipulating strings.

contains (contains(check, checkFor)): Returns true if string check contains string checkFor.

instr (instr(check, checkFor)): Returns first instead of checkFor in check, -1 if not found.

lowercase (lowercase(S is String)): Converts S to all lowercase characters.

regex (regex(regex, compareTo)): Returns true if string compartTo matches regular expression regex

replace (replace(base, checkFor, replaceWith)): Returns a string containing base with all instances of checkFor replaced with replaceWith

startsWith (startsWith(check, checkFor)): Returns true if string check starts with checkFor.

subStr (subStr(S is string, start is Number, length is Number)): Returns a substring from S, starting at position start, with length length.

trim (trim(val)): Returns a trimmed version of val.

uppercase (uppercase(S is String)): Converts S to all uppercase characters.

Thread

Shiro supports threading by the use of its Thread library. This allows your application to do multiple things at a time, and unlike many implementations of threading is relatively safe. You don't have to worry about locking or thread safety, Shiro takes care of it for you. When you create a thread, a whole new instance of the Interpreter with a fully cloned Symbol Table, Function Table and Library table is spawned. The thread runs in this instance, meaning any changes it makes to variables at any scope level will be local to the thread.

Threads can communicate with each other in a couple of ways. The obvious one is via return value, which can be queried through the thread library. Threads can also post periodic "updates" and then continue running, which allows other threads to obtain information from them. Data cannot be passed into threads after their creation -- this is the key to Shiro's thread-safety.

You create a thread by calling threadStart with a function name, or threadStartCode with a string containing some Shiro code. The second parameter to both functions is a tuple representing the new Symbol Table. You can clone the existing symbol table using getGlobal() in the Std library. Both start routines return a thread id.

You can get the result of a thread by calling threadResult with the id, but if you call it before the thread is done, Shiro throws an exception. You can check to see

if a thread is done by calling threadDone with the id. The sample below is a simple demonstration.

```
use Thread

def action() {
        return sleep(5000)      # sleep() returns the duration
}

t = threadStart(action$, getGlobal())
while !threadDone(t) {    sleep(100)    }
print threadResult(t)           # 5000
```

With a sufficiently large symbol table, threadStart can take some time to execute. It is, however, possible to pass only the necessary symbols as the second parameter if you want to get fancy -- just construct an object in the normal way. Beyond that, using threads is just as easy as it looks. If your application has to perform multiple tasks that are not dependent on the same libraries or data sources, threading is the way to go.

You can get the Id of the thread that you're current running in with the global variable GlobalThreadId, which will only be set if you are, in fact, running in a thread. Here's a brief summary of the methods available in the thread library:

getThreadUpdate(id): Gets the thread update for the thread with the supplied Id.

threadDone(threadId): Returns true if the thread with the supplied Id is finished.

threadHasUpdate(id): Returns true if the thread with the supplied Id has posted an update.

threadStart(functionName is String, symbolTable is Object): Starts a thread at the function name supplied. You can use getGlobal() to clone the existing symbol table for the second parameter, or create your own scope.

threadStartCode(shiroCode is String, symbolTable is Object): As the method above, but executes the code in shiroCode rather than explicitly calling a function.

threadResult(threadId): Returns whatever the thread returned. Exception if thread not done.

updateThreadData(id, data): Posts a thread update. Pass GlobalThreadId as the first parameter and whatever object you want to post as the second.

SQL

sqlOpen(conStr)

sqlClose(handle)

sqlExec(handle, query)

sqlQuery(handle, query)

dbOpen(path)

dbClose()

dbQ(sql)

dbExec(sql)

dbSave(id, object)

dbLoad(id)

dbDefault()

ICanSave.save(id)