The Falcon Programming Language

Showdown

Giancarlo Niccolai

— 0.8.14 —

The Falcon programming language – Showdown.

Table of contents

Showdown

This document is meant for those people who already know what a scripting language is, either because they have used some of them before or because they have studied them. This is a small guide to the key features, and how to do things efficiently in Falcon.

Basic I/O

Output can be performed using the “>>” and “>” output operators and the `print` and `printl` functions.

```
> "Hello world"
printl( "Hello world" )
```

A sequence of variables can be printed through “,”:

```
a = 100
b = "Hello world"
> b, " ", a, " times!"            // prints "Hello world 100 times!"
printl( b, " ", a, " times!" )    // same, but using printl
```

The “>>” operator and “`print`” function do the same thing, but they won't print an extra newline.

To read a line of text from the console, it's possible to use the input() function:

```
>> "Say something: "
said = input()
> "You said: ", said
```

However, for more complex work, such as input filtering, it's better to use the standard input stream class. See `stdIn()` function in the *Function Reference* for usage examples.

Output can be formatted on the fly through the “@” string expansion operator; in strings prefixed with “@”, $var and $(var:<format>) are expanded on the fly, or stored for later evaluation:

```
// On the fly evaluation:
sinVal = sin(2)  // angle in radians
> @ "Unformatted: $sinVal    -    Formatted: $(sinVal:.2)"
```

The Format class gives a greater control of on-the-fly formatting:

```
sinVal = sin(2)  // angle in radians
twoDigitNumber = Format( ".2" )
> "Unformatted: ", sinVal, "   -   Formatted: ", twoDigitNumber.format( sinVal )
```

String juggling

In Falcon, strings can be seen at the same time as both sequences (arrays) of UNICODE characters and as a pack of textual entries. Some read modes allow inputting data as binary sequences of bytes. This makes it possible to directly mangle binary data, converting it to and from text when necessary. For this reason, Falcon also provides a complete set of binary operators. Basic string operations include string concatenation, extraction, reduction and insertion:

```
str = "Hello world"
> '"',str,'" is ', str.len(), ' characters long.'

> "First five characters", str[0:5]
str[0:5] = "Goodbye"                // Goodbye substitutes Hello
> "Heading changed: ", str
str[7:] = ""                        // Remove all what follows 7
> "Just say... ", str
str += " to everybody!"             // append at the end
```

```
> "Or maybe say... ", str
```

Ranges can be pre-cached and used later:

```
str = "Hello world"
rng = [5:0]
> "First word reversed: ", str[ rng ]
```

To access the byte or UNICODE value of a certain item, the star accessor can be used. Assigning a numeric value to a string element causes that binary value to be inserted:

```
str = "Hello world"
> "First character binary value: ", str[*0]
str[0] = str[*0] | 0x20                        // binary or 32.
> "Binary old style lower-case...: ", str
```

This way to manipulate single string elements as numbers is mainly meant for binary access of byte oriented data, but it can be useful also to manipulate directly UNICODE characters.

Be sure to check out the **Memory Buffer** (MemBuf) type in the Survival Guide, which allows direct mangling of memory areas that can be used for block I/O or to exchange native data with the applications.

Strings can have also have numeric escapes, and if your editor supports one of the encodings known to the Falcon compiler and VM (UTF8 and UTF16 among the others) you can also write any character directly in that:

```
str = "Hello\x20in\x20Japanese: ようこそ！"
> str   // of course, your output media must support the VM encoding...
```

Falcon provides a quite complete set of functions to manipulate strings, so that it is also possible to perform basic string operations in functional sequences.

Arrays and dictionaries

Most of the things we have seen about string juggling is valid also on arbitrary sequences of items called arrays. Arrays are declared by assigning a list of items to a variable, like this:

```
anArray = [ one, 2, "three" ]
inspect( anArray )            // inspect() prints the contents of a variable
```

Elements in arrays are numbered from 0 to N-1, where N is the length of the array, which can be retrieved with the `len` function or method:

```
one = 1
anArray = [ one, 2, "three" ]
> "Array size: ", len( anArray )
> "Last element: ", anArray[ anArray.len() - 1 ]
```

The element number can also be negative; in that case it is counted from “the end” of the array. Using negative indices to count from the end works also on strings as well as for ranges:

```
> anArray[ -1 ]                    // prints the last element ("three")
> anArray[ - len(anArray) ]       // it's the first element (1)

inspect( anArray[ -1:-2 ] )        // last and second-last elements
inspect( anArray[ 2: -1 ] )        // from 1 (included) to last element (excluded)
inspect( anArray[ -1: 0 ] )        // whole array reversed.
```

You'll notice that if the second element of a range comes before the first element (regardless of sign), the sequence is reversed and the second element is included. Conversely, when the first element comes before the second one, the second element is excluded. This applies also to strings.

Adding an item to an array will cause the item to be appended; adding an array to another will cause the two arrays to be joined. Subtracting an array from another will cause removal of items:

```
inspect( [ "a", "b" ] + "c" )              // will be a, b, c
array = [ "a", "b" ]
array += [ "c", "d" ]
inspect( array )                           // will be a, b, c, d
array_small = array - [ "a", "d" ]
inspect( array_small )                           // will be b, c
```

Arrays can contain other arrays; for example:

```
array = [  ["a", "b"],
           ["c", "d"]  ]       // With open ( or [, lists can span multiple lines
inspect( array[1] )            // will be c, d
> array[0][1]                  // will be “b”
```

In an array declaration (as well as in function calls and in general where you have multiple expressions separated by a comma “,”) the line can be broken also without the line extender backslash “\”. Falcon parser recognizes the array declaration or function call and allows it to continue on the next line.

Dictionaries are similar to arrays, but they bind a unique key to a value:

```
dict = [ "a" => 1, "b" => 2 ]
> dict[ "a" ]                    // prints 1
dict[ "b" ] = "two"              // changes an existing value
dict[ "c" ] = 3                  // creates a new value
inspect( dict )                  // will show updated values
```

The item used as a key may be any Falcon item, including nil, a floating point number or an arbitrary class instance. The VM has a built-in mechanism that orders keys, but it is possible to redefine the ordering priority to provide a class-specific ordering sequence and equality criterion.

Trying to read an nonexistent key will cause an error; to check for the existence of a key before accessing it, the “in” operator can be used. Continuing the above example,

```
"a" in dict ? printl( " 'a' was there!") : printl( " 'a' wasn't there...")
"no" in dict ? printl( " 'no' was there!") : printl( " 'no' wasn't there...")
```

String searches in dictionaries are always case sensitive. Notice the “?:” fast if expression, borrowed from the C language; we'll describe it later.

Actually, the “in” operator is much more flexible, and can be used also to search for elements in arrays. substrings in strings and properties in objects. There is also an inverse “in” operator called `notin`.

The standard dictionary is extremely efficient to store small sets of already known data, while it is relatively inefficient when adding new items; the performance decreases as the stored items increase. Falcon provides another dictionary model which is a bit less efficient when searching for items and requires more memory to store elements, but is extremely efficient in insertion and removal; it's called `PageDict()`.

Basic math

Falcon has quite rich mathematical support. It provides a full set of binary math operators, operating with pure integer data. This is essential to deal efficiently with binary files. Binary or “||”, and “&&” xor “^^” and not “~” operators are available both in infix and self assignment form. Octal and hexadecimal number notation complete the model:

```
a = 0x80 || 0773
a ^= 0xffee
> a
```

The shift-left and shift-right operators are provided too; as the simple print-newline statement “>” must not be confused with the binary *greater than* relational operator, so the shift right “>>” is not to be confused with the simple print statement:

```
value = 0xFFFF
while value > 0
```

```
  > @"Removing a bit... $(value:X)H"
  value >>= 1    // self-assign; same as value = value >> 1
end
```

Falcon can deal with binary math up to 64 bits, and it provides the both fixed and floating point modulo operator "%". It has also a powerful built-in power operator "**" that can be used also to extract square roots. For example:

```
> "9 times 9 = ", 9 ** 2
> "Square root of 100 = ", 100 ** 0.5
```

Falcon provides standard math functions such as trigonometric functions, degree to radians conversions, float-to-integer and float math special operations (as floor, ceil, fint – floating point integer part – and so on). It has also a complete random number generator library, which is can also be used to perform random walks and extractions over collections in a blast.

Ifs and switches

Falcon provides three basic statements to control execution flow: if/elif/else, switch/case and select/case statements. The if statement is the classical statement that selects one branch depending on the truth value of an expression:

```
exp1 = 1
exp2 = 2
if exp1 == 1 and exp2 == 1
   > "1,1"
elif exp1 == 2 and exp2 == 2
   > "2,2"
else
   > "different value"
end
```

In the case there is only a statement to be executed when the check is true, the if can be shortened using the ":"

```
b = 4
if (a = (b * 2) ) > 6:  printl( "A is big enough: ", a )
```

The colon can be used to shorten many other statements, as the various loops, function declarations and in general any declaration that does not require a set of sub-statements to work. For example, switch and select statements require a list of "cases" to be also expressed inside their block, so they cannot be shortened.

A more compact "if" can be performed through the "?:" (question-colon) operator; it works like in C:

```
b = 4
a = b < 4 ? 4 : b * 2
```

If the expression at the left of the question mark (b <4) is true, the whole expression (a) gets the value of the first element after the question mark, while if it's false it gets the value of the expression following the colon.

The `switch` construct gets an expression and checks it against a series of alternatives. Alternatives can be immediate values (including nil), range of integers and in general any variable. When the value of the expression is considered equal to one of the alternatives, that code block is executed; an optional default code block can be executed if none of the alternatives matched:

```
exp = "hello"
beta = "another value"
switch exp
   case nil, "nothing"
      > "exp was nothing"
   case 1, 2, "one", "two"
      > "exp was one or two, either numeric or string"
   case 10 to 100
      > "exp was a number between 10 and 100 (included)"
   case beta
      > "exp was exactly like beta"
```

```
   default
      > "exp was something else"
end
```

If a case branch has a non-constant value, the value of the switch expression is matched against the value of the variable at the time the switch is performed. An object may redefine its equality operator so that it matches against a switched expression; for example, a regular expression instance can be used as a case branch and will be selected given matching strings. If more case branches can match the expression, the first one being declared will be used.

The select statement works exactly the same as the switch statement, but it chooses the branch depending on the item type or class name:

```
exp = "hello"
select exp
   case IntegerType
      > "exp was an integer"
   case StringType, ArrayType
      > "exp was an array or a string"
   case ParamError
      > "Exp was an object of class ParamError"
   case Error
      > "Exp was an object of class Error or of any subclass... except a ParamError"
      > "if it were a ParamError, it would have been selected before."
   default
      > "exp was something else"
end
```

Both **select** and **switch** are highly optimized when cases are all branches of the same type, while are a bit less optimized when they are all constant branches of different types and less optimized still when there is some non-constant value; however, they are always faster than an equivalent if-elif-else set.

Loops

Falcon provides a wide set of loops to iterate over certain conditions or over sequences. The most simple loop is the classical "`while`", which loops while a certain condition is true:

```
a = 0
while a < 5
  > "Looping... ", a++
end
```

This prints the numbers between 0 and 4, incrementing the loop variable after each print. When it reaches 5, the loop terminates. Another very simple looping construct is the `loop`, which is equivalent to a "`while true`"; it loops forever, until a `break` statement is encountered:

```
a = 0
loop
  > "Looping... ", a++
  if a == 5: break
end
```

The last and most powerful loop is the for/in loop, which automatically iterates over any sequence or over a range.

```
> "First 5 numbers: "
for value in [1:6]: >> value, " "
>"."
```

Notice that the loop variable assumes values in an open range [1, 6] and so only reaches '5' if the loop counts up. If it's counting down, a closed range is taken (same as for arrays and strings). A third, optional step value can be given:

> "First 5 pairs of numbers, stepping down by 2: "

```
for value in [10:2:-2]: >> value, " "
```

```
>"."
```

Inside a for/in loop, special blocks called `forfirst`, `forlast` and `formiddle` allow special processing of certain elements. The `forfirst` block is executed before the body of the first loop takes place, and the `forlast` block is executed after the last element has been processed. The `formiddle` block is executed after every element other than the last is processed, making it useful for performing actions between items. For example:

```
sequence = [ 1, "a", "b", 2 ]
for item in sequence
  forfirst
    // print an open square before the first element
    >> "[ "
  end

  // process the element (by printing it)
  print( toString( item ) )

  // prints a comma between elements...
  formiddle: >> ", "

  // and print a close square after the last element
  forlast: > " ]"
end
```

Notice that `forfirst`, `forlast` and `formiddle` blocks can be shortened with the ":" colon. The order of their declaration is purely conventional; they can be declared in any order. All the blocks are optional. Also, the unnamed item processing block (the print() call in the above example) is not mandatory.

In cases of dictionary traversal, variables for both the key and the value of each entry must be provided:

```
dict = [ "a" => "some value", "b" => "some other value" ]
for k, v in dict
   > k, " => ", v
end
```

In the case of multi-dimensional sequences multiple variables can be declared as loop variables:

```
matrix = [ [1,2,3], [4,5,6], [7,8,9] ]
for v1, v2, v3 in matrix
   > @"Element: $(v1), $(v2), $(v3)."
end
```

All the loops support both `break` and `continue` statements. Break can be used to interrupt a loop immediately; continue statement repeats the loop from the very beginning, eventually updating the current loop variable in for and for/in loops, or incrementing/decrementing the variable with given step in for loops, and finally checking the loop condition.

The for/in loop has two special statements that allow current element modify or removal. The "dot assignment" operator (`.=`) alters the current value of the item in the sequence, while `continue dropping` statement removes the current element and restarts the loop as if a `continue` were issued. The next examples removes all the odd elements and squares all the even elements in a series:

```
sequence = [ 1, 2, 3, 4, 5, 6 ]
for item in sequence
   if item % 2 == 1: continue dropping
   .= item * item
end
inspect( sequence )
```

Notice also that strings are seen as sequences, and it is possible to break them down using the for/in loop. However, if the value of each character must just be inspected, it is more efficient to access string elements through the star-access operator.

To know more about loops, look to the Survival Guide for the desired statements.

Lists and iterators

Iterators are a special language construct that allow iteration through an underlying data sequence. Iterators can be built for dictionaries, arrays, strings and lists, and they can be created either using the Iterator class constructor or the first()/last() methods of sequence items and list instances.

This example traverses a whole array using an iterator:

```
array = [ "a", "b", "c", "d", "e" ]
iter = Iterator( array )  // or iter = array.first()
while iter.hasCurrent()
   > "Current element: ", iter.value()
   iter.next()
end
```

An iterator may be created in any point of a random-access sequence (i.e. an array or a string) by providing the index of an element as the second parameter of the Iterator class constructor. The index may be negative to access the nth element from the end (as array indexes).

Dictionaries have iterators too; a dictionary iterator provides also a `key()` method which can be used to retrieve the key of an element. Consider the following example which prints all the items having a key starting with “c” letter:

```
dict = [ "alpha" => 1, "charlie" => 3, "custom" => 4, "delta" => 5 ]
iter = dictBest( dict, "c" )

while iter and iter.hasCurrent() and iter.key()[0] == "c"
   > iter.key(), " => ", iter.value()
   iter.next()
end
```

The `dictBest` function is a Falcon function that performs a fast search in the array and returns an iterator to the best match for the given key.

Lists are sequences meant for insertion and removal of items in front or at the end. They are much more efficient than arrays when it's necessary to store a dynamic set of elements which may be inserted and removed; arrays must be periodically reallocated to make room for new elements, and in case of insertion in front, all the items in the array must be shifted by one place. Lists require extra allocation for each element, and it is not possible to access a random element through the array access operator, but they are very efficient in insertion and removal. To iterate over a list, it is necessary to use an iterator, or a for/in loop:

```
lst = List( 1, 2, 3, 4, "a", "b" )
iter = lst.first()
while iter.hasCurrent()
   > "List element: ", iter.value()
   iter.next()
end

for elem in lst
   > "Again... ", elem
end
```

Iterators can be used also to erase, insert or change the value of an element, and to traverse a sequence in reverse order. For more details see the Function reference about lists and iterators.

Functions, calls and functional operators

Falcon is a heavily functional oriented language. However, it's overall structure is not strictly functional nor strictly procedural, and has a strong OOP orientation. Functional and OOP constructs live together in Falcon interacting at many levels.

Falcon's definition of any evaluable entity is “callable item”. More specifically, callable items are : functions,

methods, lambda expressions, inner functions, class constructors and callable arrays (or lists, if you prefer).

A callable item is evaluated (called) by appending open and closed parenthesis to it, containing an optional list of parameters. Some call examples are:

```
func()
func( with, some, parameter)
dataInAnArray[itemID] ( params )
obj.method()
ClassName( param1, parm2 )
[aFuncName, parm1, param2]( param3, param4 )
```

The keyword `function` declares a user defined function; the keyword `return` sets the final result of the evaluation and exits the function:

```
function myFunc( a, b )
   if a > 0
      printl( "First parameter positive" )
      return b * 2
   end

   printl( "First parameter negative" )
   return b * 3
end

a = myFunc( 2, 5 )
> a                  // will be 5*2 = 10
> myFunc( -1, 5 )    // will be 5*3 = 15
```

Functions cannot be nested, as they declare a symbol at global level; however, it is possible to nest nameless functions which are declared without an explicit name and must be immediately assigned to local symbols. In the following example we also show that it is possible to return any callable item, including a named function.

```
function myFunc( a )
   if a > 0
      inner = function( b )
         return b * 2
      end
   else
      inner = function( b )
         return b * 3
      end
   end

   // let's return the selected function
   return inner
end

// select one of the functions
eval_func = myFunc( 1 )
> eval_func( 5 )               // will be 10

// or a direct call of calls
> myFunc( -1 ) ( 5 )           // will be 15
```

Lambda expressions are functionally equivalent to nameless functions, with the exception that they are composed of only one parametric expression whose evaluation result is immediately returned. So, for simple calculations, or even complex sequences of functional constructs as `iff` (functional if) and `cascade` they are more compact.

For example, this is the same as above written with functional constructs:

```
function myFunc( a )
   return a > 0 ? lambda b => b * 2  :  lambda b => b * 3
end
```

```
// select one of the functions
eval_func = myFunc( 1 )
> eval_func( 5 )    // will be 10

// or a direct call of calls
> myFunc( -1 ) ( 5 )     // will be 15
```

Eleven lines in one!

Nameless functions and lambda expressions support the functional semantics of closure. They can access parent's local symbols and store them as they are in the moment they are called for later evaluation.

For example:

```
function makeMultiplier( operand )
   return lambda a => operand * a
end

mult2 = makeMultiplier( 2 )
> mult2( 10 )  // will be 20
```

Inner functions are nameless functions that don't respect closure; they re-protect their symbol table as if they were top-level functions. In this way, they provide declaration scope protection.

A functional construct similar to the "?" operator is the `choice()` function, which returns the second parameter if the first evaluates to true, and the third parameter if the first evaluates to false.

```
a = 1
> choice( a > 0, lambda b => b * 2,  lambda b => b * 3 )( 5 )  // will be 10
```

Falcon supports variable parameter call convention; this means that it's possible to call a callable item with less or more parameters than the ones the item is expecting. When calling with more parameters, the extra one will be ignored; when calling with less, the missing ones will be filled with "nil". For example:

```
function showParams( p1, p2, p3 )
   > "P1: ", p1, "\nP2: ", p2, "\nP3: ", p3
   > "------------"
end

showParams(); showParams( "Just one" )
showParams( 1, "two" ); showParams( 1, nil, "three" ); showParams( 1, 2, 3, "four" )
```

Notice, you can use ";" to write more than one statement on one line.

The "|" (pipe) operator creates a "future binding", which can be used to send specific parameters by naming them, ignoring their order. For example (continuing the code above):

```
showParams( p2|"Second parameter passed by name")
bound = p3 | "Third parameter passed by name"
showParams( bound )
```

The function lbind can create bindings dynamically:

```
bound = lbind( "p2", "Dynamic second parameter" )
showParams( bound )
```

The core functions paramNumber() and paramCount() allow retrieval the nth parameter (zero based):

```
function showParams( mandatory )
   > "Mandatory: ", mandatory
   for i in [ 1 : paramCount() ]
      > "P", i+1, ": ",  paramNumber( i )
   end
   > "------------------"
```

```
end

showParams( 1, "two" ); showParams( 1, nil, "three" ); showParams( 1, 2, 3, "four" )
```

In the "symbol aliasing" section we'll talk about *by reference* parameters passing.

Please, notice that function declarations are considered module globals; they can be used either before or after their definition; for example:

```
> postCall()
function postCall()
   return "Declared after usage"
end
```

Declaration position is just a matter of taste.

An array whose first element is a callable item is itself callable, and the elements before the first one are sent to the callable item as parameters:

```
function showParams()
   for i in [0 : paramCount()]
      > "P", i+1, ": ",  paramNumber( i )
   end
   > "------------------"
end

[showParams, "Just one"]()
[showParams, 1, "two" ]()
[showParams, 1, nil ]( "three" )
[showParams, 1, 2 ]( 3, "four" )
```

In this way it is possible to “store” both a function and its parameters as a complete deferred call.

In functional context, a sequence (arrays) whose first element is callable is dubbed “Sigma”, and gets evaluated by calling it and substituting it with its return value. All other language elements are called “Alphas” and they are never evaluated. For example, look at the following:

```
test = lambda a => a > 100
iff( [test, 101], [ printl, "Test passed"],  [printl, "Test failed"] )
```

The `iff` functional construct evaluates the first element; it's an evaluable so it gets called, and it evaluates to true (101 > 100). So, the second element is evaluated; it's an evaluable too, so it gets called and the output line gets printed. Compare this with immediate calls:

```
test = lambda a => a > 100
iff( test( 101 ), printl( "Test passed" ),  printl( "Test failed" ) )
```

All the immediate calls are evaluated before the outer `iff` call. This means that both the printl calls will be called before `iff` has a chance to see that test(101) has returned “true”.

It is possible to achieve a little more functional-like expression, and removing a lot of ugly commas, using the functional-style sequence declaration ".[". The `iff` call we have seen can be rewritten as:

```
.[iff .[test 101] .[printl "Test passed"] .[printl "Test failed"]]()
```

Last but not least, a method can take part in functional constructs like shown here:

```
object testObj
   function aMethod( val )
       > "Method being called with: ", val
   end
end

test = lambda a => a > 100
iff( test( 101 ),  [testObj.aMethod, "Ok"], [testObj.aMethod, "Failed"] )
```

We'll see more about classes, objects and methods later on. For more details about functional operators and functional evaluation, see the Function reference at "*Functional programming support*" chapter.

Item aliasing

Each item in Falcon can be both statically and dynamically aliased. A static aliasing, also called "reference" can be obtained with the "$" operator. Any assignment or change in the aliased variable is reflected in both instances:

```
a = 0;  b = $a      // notice the ";" separating two statements on the same line
> b                 // is 0
a++; >b             // is 1
b++; >a             // is 2
```

A reference can be broken with "$0"; assigning $0 to an alias, the aliased variable is freed and the alias becomes nil. Dynamic alias is performed through the "indirect" # operator:

```
aString = "Hello world"
alias = "aString"
> #alias              // Prints "Hello world"
```

Exactly like the string expansion operator, the indirect operator can be applied more than once in a row:

```
v1 = "Hello";  v2 = " ";   v3 = "world"
alias = "var"
for i in [1:4]
   var = "v" + toString(i)
   >> ##alias
end
printl()
```

The first aliasing resolves to "var", which contains "v1", "v2" and "v3" names in each iteration.

The indirect operator can come quite useful also in functional programming; the following code slice selects one of two functions:

```
function f0(): > "Instance of f0"
function f1(): > "Instance of f1"
function selector( value )
   val = "f" + (value %2 ? "0": "1")
   return #val
end

selector(100)()  // prints "Instance of f1"
selector(101)()  // prints "Instance of f0"
```

The indirect operator can be applied also to vector and object accesses (mixed in any order):

```
class test( name )
   name = name
end
vector = [ test( "Hello"), test(" "), test("world") ]

for i = 0 to 2
   alias = "vector[" + toString(i) + "].name"
   >> #alias
end
printl()
```

This indirect access has two limitations. First, it cannot evaluate expressions nor call functions (however, notice that it can resolve in a callable item that can then be called). Second, at the moment the access is read only.

The static aliasing (reference) can be used to provide a function or a method with by-ref parameters that can be modified during the call:

```
function changer( data )
   data ++
end

a = 0
changer( a )          // pass by value
> a                   // it's still 0
changer( $a )         // pass by reference
> a                   // is 1!
```

Objects and classes

Falcon provides singleton objects and classes from which multiple instances can be derived. They have members, which may be properties (if they contain data) or methods (if they refer to a function or to a callable item). They have also an inheritance tree and can inherit from multiple ancestors, and they can be given attributes.

An attribute is a binary property (an instance may have it or not) that can be given to any instance of any class, or to any object. Objects and instances having a determined attribute form an attributed set, which can be inspected or invoked as a whole through “attribute broadcasts”.

A class is declared through the class keyword. Properties must be declared assigning them an initial value; then, an `init` block may optionally be provided to initialize the instances and then a set of functions can be declared. A final “`has`” clause defines attributes that a class instance should have by default:

```
attributes: isVisible

class WorldItem( param1, param2 )
   name = param1
   extra = param2
   property = nil
   otherProp = 0

   init
      if param2 == nil: self.extra = "Not given"
   end

   function whoAmI()
      return self.name
   end

   has isVisible
end
```

Both the property initialization and the init block can access the instance parameters, which are given at instance creation. To create an instance, the class must be “called” as if it were a function:

```
inst = WorldItem( "a name" )
```

To initialize a property, any expression may be used. This includes function or other property values; most notably, method names can be used. In the following example, the `alias` property is a copy of the `whoAmI` method; they can be used interchangeably:

```
class WorldItem( param1, param2 )
   name = param1
   extra = (param2 == nil? "not given" : param2 )
   alias = self.whoAmI

   function whoAmI()
      return self.name
   end
end
```

Inheritance is expressed using the from clause. Base classes may be initialized by passing them parameters or constant values:

```
class WithAName( pn )
   name = pn
end

class WithAnExtra( pn )
   extra = (param2 = nil? "not given" : param2 )
end

class WorldItem( param1, param2 ) from \
        WithAName( param1 ), \
        WithAnExtra( "I am a world item!" )
    //...
end
```

Name scope protection is not provided; all the members are public. It is possible to overload properties or method by redefining them in the subclasses, and it is possible to access to some parent's method using the class name of the ancestor the method comes from:

```
class Base
   function myName(): return "I am a base item"
end

class Derived from Base
   function myName(): return "I am a derived item"
end

instance = Derived()
> instance.myName()      // derived
> instance.Base.myName() // base
```

Classes can be cached as any other callable symbol; it is possible to assign a class to an arbitrary item and then call that item; the return value will be a new configured instance. Continuing the above example:

```
//...
dv = Derived
instance = dv()
> instance.myName()      // derived
```

Singleton objects are globally visible special items. An object declaration is equivalent to declaring a class with the same name, and deriving an element from it; however, this is done during link time, and objects get readied (initialized) before the main script is executed. In this way, objects referenced by other modules being linked one after another are ready to be used as the program grows, before it becomes fully functional and ready to run. Objects can be derived from base classes, but they cannot have initialization parameters and nothing can be derived from them.

```
class Base1( ival )
   name = ival
end

class Base2( ival )
   descr = ival
end

object Myself from Base1( "Myself" ), Base2("Call me... a programmer")
   init
      > "Myself object being initialized..."
   end

   function myName(): return self.name
end
```

In short, object declaration is exactly like class declaration, except for the fact that the object name after the `object` keyword cannot be followed by a parameter list.

A set of core functions and methods allow the determination of the base class and the class name of instances; search for "`baseClass`" method in the Function Reference.

Other than the `self` object, which allows access properties and method in the currently called instance, the Falcon VM provides a `sender` object which stores the object from which a call was issued. If the call was performed from the main script or from a function which is not a method, the sender object is nil. For example:

```
object saved
   name = "This item..."
   function toSave(): saveMe()
end

function saveMe(): > "I am saving ", sender.name

saved.toSave()
```

An interesting characteristic of the Falcon OOP model is that it is possible to assign any function (or callable item) to a property and make it a method. The self object will assume the value of nil if a function is not called as an object method, while it will assume the value of the object owning the called method in normal method calls. For further readings, see the *Survival Guide* about objects and classes.

Attributed sets and broadcasting

Objects and instances being given some attributes enter a so-called "attributed set", which is a list containing all the items which have received a certain attribute. Iterating over an attribute with a for/in loop or creating an iterator out of an attribute, it is possible to inspect all the objects in the attributed set. For example:

```
attributes: bset

class tester( name )
   name = name
end
// create 4 instances the fast way...
t1, t2, t3, t4 = map( tester, [ "one", "two", "three", "four" ] )

// give the attribute to some of the instances...
give bset to t1, t2

// iterate over the attributed set for bset
for obj in bset
   > "Instance in set: ", obj.name
end
```

or changing the above for/in loop:

```
iter = bset.first()
while iter.hasCurrent()
   > "In set: ", iter.value().name
   iter.next()
end
```

One interesting usage of attributes is to generate "broadcast" callbacks on all the instances in the attributed set. The function broadcast() takes a single attribute or an attribute array as first parameter; for all the attributed sets defined on the given attributes, a method being named after the attribute is searched in each instance, and if found, is called with the remaining parameters of the broadcast function. For example:

```
attributes: cbSubscription

object test
```

```
   function cbSubscription( param )
      > "Responding to broadcast: ", param
      return false // returning true will break the chain
   end

   has cbSubscription
end

broadcast( cbSubscription, "The original message" )
```

If the instances having an attribute do not provide a callable item with the same name as the attribute being broadcast, they are simply skipped.

A small set of powerful functions and methods allow operations on the attributes and on the attributed set. For more details, see the "*Attribute paradigm support*" chapter in the Function Reference.

Event marshaling

Message broadcasts reaching an instance may have any parameter. However, Falcon recognizes special values to messages called "events", which are arrays having a string as the first element (the event name). Falcon provides a small set of special functions that allow automatic dispatching of broadcast events to different handlers in objects. Look at the following example:

```
attributes: cbEvent
class EvtReceiver
   cbEvent = [marshalCBX, "on_", false ]
   has cbEvent
end
```

Now, every class or object derived from `EvtReceiver` will be able to dispatch automatically an event to a method called "`on_<event name>`" in the instance. For example:

```
object Receiver from EvtReceiver
   function on_SomethingHappened( param )
      > "Hey, ", param, " happened!"
   end
end

broadcast( cbEvent, ["SomethingHappened", "this event" ] )
```

To know more about marshaling, consult the Survival Guide on "Attributes and Messages" and read about the "marshalCB" and other indirect execution functions in the Function Reference.

Prototype OOP

Since version 0.8.12, Falcon has full support for prototype based Object Oriented Programming. It is possible to create a prototyped instance from either a dictionary or an array; dictionaries can be "blessed" to become instances, while arrays provide the concept of "bindings", that is, dynamic assignment of data to accessible properties.

```
inst = bless([
   "_prop" => 0,
   "inc" => function(); self._prop ++; end,
   "dec" => function(); self["_prop"]--; end,
   "value" => function(); return self._prop; end
  ])

inst.inc(); inst.inc(); inst.dec()   // just to show access with [] and dot
> inst.value()                       // shall be 1
```

In the above example, we create and bless a dictionary on the fly. As shown, the dictionary can still be seen as a

dictionary, but it provides a self item which is both accessible via the dot accessor and via the dictionary subscript accessor.

Array bindings take the concept a step further and allow reflection of the bound property into its structure. For example:

```
call = .[printl &value '.']
call.value = "Hello world"
call()
```

The "&" symbol creates a "late binding", which is resolved into the value of the bound data during calls and functional evaluations. The special &self binding resolves in the array itself that can then be passed to the evaluated sequence:

```
eval( .[inspect &self] )
```

will cause inspect to receive the same array that was evaluated.

Both blessed dictionaries and bindings can receive externally written methods:

```
function checkMe( prompt )
  > prompt
  for item in self: > item
end

val = [ 'an', 'array' ]
val.do = checkMe
val.do( "The val array is so structured:" )
```

In exactly the same way as "future bindings", array late bindings can be cached into any Falcon variable, and they can be created dynamically by the lbind function.

```
bind1 = &v1
bind2 = lbind( "v2" )

call = .[ (function( p1, p2 ); > "v1:", p1; > "v2:", p2; end) bind1 bind2 ]

call.v1 = "first value"
call.v2 = "second value"
call()
```

As shown in the example, the bind1 and bind2 variables are interpreted as "&v1" and "&v2", which are resolved using the array bindings assigned in call.v1 and call.v2.

Template files

Falcon provides a "template file" mode that is designed to output an expanded input, not unlike dynamic web oriented languages as ASP and PHP. The escape sequence "<?" or "<?fal" escapes to the Falcon parser. With respect to those languages, Falcon presents three relevant differences. First, the compiler creates a Falcon binary module out of the template file, which can then be processed through normal VM output. In other words, embedded applications can take advantage of template files by intercepting the VM streams, and it is possible to pre-compile active pages and store them as FAM modules directly at the server site to cut compilation time.

Second, it is possible to enter and exit from escaped code at any level; for example, if wide sections of text or wholly or partially un-escaped code is to be generated inside a loop in a function, it is possible to simply exit the escaped code and enter the plain template text.

Third, it is possible to seamlessly merge pure Falcon modules, template documents and binary modules into a completely integrated template-based application.

In this example, procedural and escape based templates are seamlessly merged:

```
<HTML>
<HEAD><TITLE>Hello world</TITLE></HEAD>
```

```
<BODY>
   <H1>Parameters:</H1>
   <TABLE><TR><TH>Id</TH><TH>Name</TH></TR>
   <? // We'll echo the parameters of this script in a table
      say_parameters()
   ?>
   </TABLE>
</BODY>
</HTML>
<? // we'll see a sample of template procedural generation first ...
function say_parameters()
   count = 1
   for elem in args
     print( "<TR>" )
     say_line( count++, elem )
     printl( "</TR>" )
   end
end
//... and then a sample of reverse template
function say_line( id, name )
   // here we exit the code and re-enter the templating.
   // we'll re-escape again to print variables
?>  <TD><?= id ?></TD><TD><? print( name ) ?></TD><?
   // and here we can close the function.
end ?>
```

The effect of this test can be seen saving this script as `test.ftd` and running it with the `falcon` command line tool.

Meta compilation

Falcon provides a virtual machine that can be directly used by the script compiler. In other words, it is possible to program the compiler and its output directly from a complete inner script. The `\[ ... \]` escape can contain a "meta script", which can include also references to external modules, and everything that is delivered to the standard output stream in that section is fed into the compiler at that position in the host script.

For example:

```
formula = \[
   if SIDE == "left"
      >> "sin"
   else
      >> "cos"
   end
\]( angle )

// or more compactly
formula = \[ >> (SIDE == "left" ? "sin":"cos") \]( angle )
```

This will compile into either one or the other version depending on the `SIDE` variable (or constant) setting.

The meta compiler also provides a `macro` command which makes the task of writing meta-scripts a bit simpler:

```
macro square(x) (($x * $x))
> "9 square is: ", \\square(9)
```

Macros can contain arbitrary code; for example they can create classes:

```
macro makeClassWithProp( name, property ) (
   class $name
      $property = nil
   end
```

```
)

\\makeClassWithProp( one, speed )
\\makeClassWithProp( two, weight )

inst_one = one()
inst_two = two()

> "Instance of class one: "
inspect( inst_one )

> "Instance of class two: "
inspect( inst_two )
```

Macros are actually syntactic sugar for meta-compilation; having defined the macro like in the last example, it is possible to produce a set of classes with this meta-code:

```
\[
   for clname in [ "one", "two", "three", "four" ]
       makeClassWithProp( clname, "property" )
   end
\]
```

Conclusions

This small guide to the most interesting features of Falcon is by no means exhaustive. To know more about Falcon, consult the Survival Guide. A complete Function Reference is always available and updated at each release.

Important aspects of Falcon that this brief document didn't touch are the modular design (`load` and `export` directives), the reflexive complier and module interfaces, and the standard modules for regular expression support, the configuration parser etc.

Other than simply using Falcon, it is possible to extend it and provide new functionality; detailed instructions are provided in the module writer's manual.

The Embedding Manual shows how to integrate the Falcon scripting engine into other applications in a step-by-step example driven easy to follow guide.

Visit The Falcon Programming Language home at http://falconpl.org, and read the online documentation, which includes user maintained wiki guides.