



Published on Free Software Magazine (<http://www.freesoftwaremagazine.com>)

The Falcon Programming Language: a brief tutorial

A comprehensive overview of a programming language designed as a mixture of programming styles.

By [Giancarlo Niccolai](#), [Dennis Clarke](#)

PUBLISHED: Recently published

The Falcon Programming Language is a typeless language born for rapid development, prototyping, and ready-made integration. We may also describe Falcon as a “scripting” language with features that enable the programmer to create even complex multi-threaded applications. It mixes several different programming paradigms into an unique blend of constructs, overcoming the limitations and partialities of other languages.

The objective of this brief article is to be very practical with code examples as well as step by step simple instructions. I won't reprise the historical and technical details that led us to developing this new programming language. Please feel free to visit the main Falcon Programming Language site <http://www.falconpl.org> for plenty of details.

The Falcon p. l. presents multiple programming paradigms with a unique blend of interconnected concepts. Procedural, functional, both prototype and class based-object oriented, message oriented and tabular programming. Each of these extending into the others gives you the freedom to choose the perfect blend to represent the problem at hand. Falcon allows you, the programmer and the real human, to overcome the limitations of many programming languages that force you to restrict your thinking. Falcon allows you to just follow the flow of your thoughts. “Apply Mind over matter” is the motto for Falcon. It is *not* the programmer's mind that must reduce down to the simplified representation of the design, forced by single-minded programming languages, regardless of how mathematically elegant those representations may be. It is the language that must cope with the complexities of a non-linear thought. Falcon gives human thought the means to express itself via the computer paradigm. We feel that Falcon may be the cure for the common computer.

The Basics

Falcon has native strings, numbers, arrays and dictionaries. Falcon has a set of single line and block (multi line) imperative statements. Since version 0.8.12, Falcon provides an experimental interactive mode, useful for fast tests and for learning the expressions in this new language. We can jump right in with a tutorial with the command “falcon -i”. You are now in interactive mode where the following code can be entered:

```
array = ["Have", "a", "nice", "day"]           // (1)
for elem in array                             // (2)
    >> elem                                    // (3)
    formiddle                                  // (4)
```

The Falcon Programming Language: a brief tutorial

```
    >> " "  
end  
  
forlast  
  > "!"                               // (5)  
end  
end
```

Here is what you will see on the command line of your OS (Solaris is used in this example):

```
bash-3.2$ falcon -i  
Falcon compiler and interpreter.  
Version 0.8.14.2 (Vulture)  
  
Welcome to Falcon interactive mode.  
Write statements directly at the prompt; when finished press CTRL+D to exit  
>>>  
>>> array = ["Have", "a", "nice", "day"]  
: Array  
>>> for elem in array  
... >> elem  
... formiddle  
... >> " "  
... end  
... forlast  
... > "!"  
... end  
... end  
Have a nice day!  
>>>
```

We have just created an array of strings in line (1). Then we iterate (2) over the list taking one element at a time and printing on the standard output (3). The double greater-than symbol “>>” prints the given item to the standard output; the single “>” will add also a new-line.

The `for/in` block is divided into four regions, three of which are optional; the `forfirst` area (executed when the loop is started and just once), the main area (always executed), the `formiddle` area (executed after the main area for all the elements except the last) and the `forlast` area (executed after the main area for the last element has completed).

All Falcon block statements can be shortened in case they contain just one statement through the “:” symbol. So, we may re-write the above loop in a more compact way (and adding a `forfirst` block) :

```
for elem in array  
  forfirst: >> "Contents of the array: "  
  >> elem  
  formiddle: >> " "  
  forlast: > "!"  
end
```

Here is what you will see on the command line of your OS (Solaris again):

```
bash-3.2$ falcon -i  
  
Falcon compiler and interpreter.  
Version 0.8.14.2 (Vulture)  
  
Welcome to Falcon interactive mode.
```

The Falcon Programming Language: a brief tutorial

```
Write statements directly at the prompt; when finished press CTRL+D to exit
>>> array = ["Have", "a", "nice", "day"]
: Array
>>> for elem in array
... forfirst: >> "Contents of the array: "
... >> elem
... formiddle: >> " "
... forlast: > "!"
... end
Contents of the array: Have a nice day!
>>>
```

As in the vast majority of modern languages, functions can be called by applying parenthesis to a symbol; `print` and `println` are two functions equivalent to “>>” and “>”, fast-print operators respectively, we can write:

```
for elem in array
  forfirst: print( "Contents of the array: " )
  print(elem)
  formiddle: print( " " )
  forlast: println( "!" )
end
```

But a very important working principle of Falcon is that computational units (functions) are actually expressions; as such, we can be a bit fancy and pick the best function to employ as the result of an expression:

```
for i in [ 0 : array.len() ] // (1)

  /* A bit naive, but this is just a sample */
  if i == array.len() - 1 // (2)
    separator = "!"
  else
    separator = " "
  end

  // (3)
  ( i != array.len()-1 ? print : println ) \
    ( array[i], separator )
end
```

In (1), the code performs a loop in a “range”, declared as `[n:m]`; the `i` variable gets the values between `n` and `m-1` (one less than `m`). In (2) there is a simple “if” statement in which we assign a space or a terminal character to the variable “separator”. This simply depends on the value of “i” in this loop and clearly we are looking for the last element in the array. Then, we see the principle at work in (3). The *ternary if*, `<a> ? : <c>` expression can either resolve in the “print” (if `i` is not `array.len()`) or `println` value, which is immediately called with two parameters: the `i`-th element in the array and the separator. Both functions print all the parameters they receive, and so we see another important principle of Falcon: computational units can be called with an arbitrary number of parameters. Parameter binding is resolved (efficiently) at runtime, and many functions are prepared to receive different parameter sequences.

Falcon also has a wide set of “standard” statements:

1. the “while” loop,
2. if-elif-else,
3. break/continue statements in loops,
4. a powerful “switch/case” statement
5. and many more ...

We are skipping them right now, as they do nothing unexpected or extraordinary (well, not quite; many, as the “continue dropping” statement, have received some power ups in Falcon). We’ll see some of the most interesting and peculiar statements as we continue the exposition of the conceptual Falcon framework.

Callables & bindings

A less formal, but convenient, term to refer to “computational units” in Falcon is *Callable*. Falcon *callables* can be divided in two classes: direct and indirect. The first ones are items referring directly to computational units; the second are callables storing parameters. For example:

```
println( "Hello", " ", "world" )           //(1)
fcall = [println, "Hello", " ", "world"]
fcall()                                     //(2)
```

The call in (1) is direct, while `println` is called indirectly with its parameters in (2).

As you can imagine, an array declaration is also an expression and as we can call expressions we can easily print our “hello world” thus (from the interactive shell):

```
>>> [println, "Hello", " "]( "world" )
Hello world
```

The “Have a nice day!” loop can compose the sequence to be called and then call it. Here we can see this in action on a Solaris machine with the Falcon package from [Blastwave.org] (<http://www.blastwave.org>)

```
bash-3.2$ falcon -i
Falcon compiler and interpreter.
Version 0.8.14.2 (Vulture)

Welcome to Falcon interactive mode.
Write statements directly at the prompt; when finished press CTRL+D to exit
>>> array = ["Have", "a", "nice", "day"]
: Array
>>> fcall = [ print, nil, " " ]
: Array
>>> for i in [ 0 : array.len() ]
...   if i == array.len()-1
...     fcall[0] = println           //(1)
...     fcall[2] = "!"
...   end
...
...   fcall[1] = array[i]           //(2)
...   fcall()
... end
Have a nice day!
>>>
```

The program uses the default call modifying the central element (2), but in the last loop it also changes the surroundings. However, in complex expressions it may be hard to track the index of a call that must be changed. In fact, array elements can also be known by name, through the late bindings:

```
array = ["Have", "a", "nice", "day"]
fcall = [ print, &word, &separator ]    //(1)

// prepare the default
```

The Falcon Programming Language: a brief tutorial

```
fcall.separator = " " // (2)

for i in [ 0 : array.len() ]

  if i == array.len()-1
    fcall[0] = printl
    fcall.separator = "!"
  end

  fcall.word = array[i]
  fcall()
end
```

The `&` operator (1) creates a *late binding*, which can be assigned a dynamic value via the `.` dot accessor (2). Late bindings are language items, and can be assigned dynamically, and the dot accessor does not need to be applied on an already existing binding. The following code works on a different principle, switching in the desired binding at the right time.

```
fcall = .[ printl "I am " nil ] // (1)
fcall.happy = "happy :-) !"
fcall.sad = "sad... :-("

for count in [0:5]
  fcall[2] = random( &happy, &sad ) // (2)
  fcall()
end
```

With a test run:

```
I am happy :-) !
I am happy :-) !
I am sad... :- (
I am sad... :- (
I am sad... :- (
```

In (1) notice the `.` [*comma-less array declaration*; it works exactly as the other declarations seen so far, but tells the compiler that commas around elements are optional. The random function (2) selects (randomly) one of the late bindings matching the name of a previously assigned value. In fact, there is also a function called `lbind()` which creates a late binding by name; so (2) is equivalent to:

```
fcall[2] = lbind( random( "happy", "sad" ) )
```

String indirects and expansions

Composing dynamically late binding names is quite a powerful feature. It is similar to the ability to compose dynamically arbitrary symbol names with the indirect operator `#`; see an example from the interactive compiler:

```
>>> data = [ "cross", "head" ]
: Array
>>> for count in [0:5]
...   d = # random( "data[0]", "data[1]" ) // (1)
...   printl( "Launch " + count + ": ", d ) // (2)
... end
Launch 0: head
Launch 1: cross
```

The Falcon Programming Language: a brief tutorial

```
Launch 2: head
Launch 3: cross
Launch 4: cross
>>>
```

The # indirect expansion operator can be applied directly to symbols, array accessors as in (1), dot accessors or any arbitrary sequence of symbols, square and dot accessors. So, the effect of (1) is that of evaluating either the element 0 or 1 in the symbol called “data” and returning it into d.

The same expansions are available inline to strings via the “@” string expansion operator. We can rewrite (2) in a more comprehensible form, which expands the elements following the dollar marker “\$” into their value:

```
> @ "Launch $(count): $d"
```

You’ll remember the fast print operator (“>”) from the first example. The parenthesis around `count` are necessary to disambiguate the surroundings (the colon has a special meaning), and are optional. We can therefore rewrite the above program thus:

```
data = [ "cross", "head" ]

for count in [0:5]
  id = random( 0, 1 )
  > @@ "Launch $(count): $$ (data[ $id ])" // (1)
end
```

What is happening here? — In (1), the first application of @ expands the items prefixed with a single dollar sign into their current local value, substitutes \$\$ into \$ and returns the composed string. Now we have something like `Launch 0: $(data[1])`, which is expanded again by the other @.

Notice also that string expansion can also apply string formatting on the fly. Let us therefore change (1) with the following code:

```
> @@ "Launch $(count): $$ (data[ $id ]:r6)"
```

that is, adding “:r6” inside the last parenthesis pair, Falcon applies the “r6” format to the expanded string. This means to right justify the expanded string in a fixed field size of 6 characters, like in the following sample run:

```
Launch 0:   head
Launch 1:  cross
Launch 2:   head
Launch 3:  cross
Launch 4:   head
```

As a final statement, format codes are managed by the `Format` class, which can work similarly to the formatter classes in the Java SDK.

Defining functions

Falcon has many ways to define a function. A procedural-like approach is the “function” block statement:

```
function hail( name )
  > "Hello ", name, "!"
end
```

The Falcon Programming Language: a brief tutorial

See an example run in the command line interpreter:

```
>>> function hail( name )
... > "Hello ", name, "!"
... end
>>> for name in .[ 'Tom' 'Ed' 'Sam' ]: hail(name)
Hello Tom!
Hello Ed!
Hello Sam!
>>>
```

Notice the combination of `. [comma-less declaration` and the usage of single quotes. In fact, double quotes strings are merged when they are separated only by white spaces or newlines, so

```
.[ "Tom" "Ed" "Sam" ]
```

and

```
.[ "TomEdSam" ]
```

are the same. To separate strings it's also possible to use the dot-string notation `. "`, or use the comma to separate elements, or to use the international string marker `i "`. This feature make it possible to internationalize programs easily through an integrated translation tool.

We know that every function can be called with a variable set of parameters, regardless of its declaration. Thus:

```
function hail( name, age )
  if age == nil
    > "Hello ", name, "!"
  elif age > 0 and age < 12 // (1)
    > "Hi little ", name, "!"
  elif age <= 18
    > "Yo ", name, "!"
  elif age <= 150
    > "Good morning, Mr. ", name, "."
  else
    > "Ehm, you don't have a proper age."
  end
end

// Let's call this function with two parameters...

for name, age in .[ .['Ed' 10] .['Sam' 15]
                  .['Smith', 30] ] // (2)
  hail( name, age )
end

// and also call it with just one parameter
hail( "Mark" ) // (3)
```

Notice in (2) that the `for/in` loop can also expand multiple variables, provided that the elements in the lists have all the same number of elements (as in this case). Also, see in (1) that the `>` sign is normally used to compare the value of items, and the `and` operator binds logic expressions. In fact `>` is considered a fast print only if found at the beginning of a line.

The Falcon Programming Language: a brief tutorial

In (3) the function `hail` is called with just one parameter, and Falcon doesn't complain; instead, it sets "age" to nil and proceeds. The program can determine this fact and take proper actions.

The function keyword can be used to declare globally-visible functions only at top level, but as expected from a functional language, it is also possible to define functions on the fly, and then consider them to be expressions. As expressions, we can assign them to variables. We can use them as parameters for other functions or call them directly.

There are three ways to create functions locally (and possibly nested). Lambda expressions are parametric expressions, which can be seen as one-expression-only functions. They work like this:

```
inspect( map( lambda x => x * 2, [ 2 3 4 5 ] ) )
```

The `lambda x => x * 2` expression says that our parametric expression receives one parameter and returns it times 2. The `map` function takes the computational unit passed as first parameter and creates an array by applying it to all the items in the array it receives as second parameter. The `inspect()` function is a useful debug function traversing and displaying the contents of deep items.

A lengthy list of functions within Falcon may be seen at:http://www.falconpl.org/project_docs/core/functions_list.html

Out of curiosity, you may see a lambda in action by itself (from the interactive interpreter):

```
>>> (lambda x,y => x*y)(2, 3)
: 6
>>> mul = lambda x, y => x * y
: Function _lambda#_id_2
>>> mul( 4, 5 )
: 20
```

The function keyword followed directly by parameter declaration can be used to create a more complex callable:

```
>>> absmul = function( x, y )
...   n = x * y
...   if n < 0: return -n
...   return n
... end
: Function _lambda#_id_8
>>> absmul( 2, 3 )
: 6
>>> absmul( 2, -3 )
: 6
>>>
```

The "innerfunc" keyword works as nameless function, but it doesn't perform a closure. This means that lambda expressions and nameless functions can remember the variables set by their parent and use them later on (this is called closure), while innerfunc grants a complete local variable scope protection. In the following example we create a multiplier function by closing the parameter in the parent creating it:

```
>>> function makeMultiplier( value )
...   return lambda x => x * value
... end
>>> multBy2 = makeMultiplier( 2 )
: Array
```

The Falcon Programming Language: a brief tutorial

```
>>> divBy2 = makeMultiplier( 0.5 )
: Array
>>> multBy2(2)
: 4
>>> divBy2(2)
: 1
```

This happens because the returned lambda remembers the `value` variable. The `function` keyword does that also:

```
>>> function rememberMe( value )
...   return function( prompt )
...     if prompt
...       return prompt + " " + value    //(1)
...     end
...   return value
... end
... end
>>> func = rememberMe( "Hello" )
: Array
>>> func()
: Hello
>>> func( "::::" ) //(2)
: :::: Hello
```

In (1) we check if `prompt` is a true value; the `nil` value is always considered false, so we'll return a string with a prompt prefixed with the closed "value" variable only if `prompt` is not `nil`, as in (2).

It may be interesting to inspect the contents of `func` now; calling `inspect(func)`, you'll see that the item is actually an array composed of the actual code and its closed parameters:

```
Array[2]{
  Function _lambda#_id_15
  "Hello"
}
```

When the Falcon compiler finds a closure it creates a prepended ghost parameter and stores the actual value of the closed variable in an array that can then be treated and called as a normal function. However, if we need to, we can still access the closed variables and change them:

```
func[1] = "Hello again"
func()
```

Talking about closure, let me just write a note on Falcon scoping. Global variables are normally accessible by every level of the program. Local variables are visible only in functions declaring them (and in their immediate nested children lambda and functions) while `innerfunc` declaration provides full isolation and local scoping:

```
val = "A global value"

function test()
  > val                                //(1)
  k = 1
  return innerfunc( k )
    > k * 2                             //(2)
  end
end
```

The Falcon Programming Language: a brief tutorial

```
f = test()
f( 2 ) // (3)
```

In (1) we get the value of the global variable, (2) will use its own version of “k” rather than using the owner local variable through closure, and (3) will just print 4.

However, assigning a variable in any scope declares the variable as local of that scope. To alter the value of a global variable, or to read it in a place where the variable has still not been assigned, the global keyword can be used:

```
function test()
  global val
  val = "Set by test()"
end

test()
> val // (1)
```

As can be seen in (1), we’re using a global variable created from inside test().

Call conventions

Talking about functions, Falcon has three primary calling conventions. They are named parameters, pass-by-reference and variable parameter conventions.

Suppose we have a function inspecting its parameters:

```
function whatParams( alpha, beta, gamma )
  > "Alpha was: ", alpha
  > "Beta was : ", beta
  > "Gamma was: ", gamma
end
```

To send only beta, we may nil alpha:

```
whatParams( nil, "Hello" )
```

Or we may tell exactly what to send:

```
whatParams( beta | "Hello" )
```

The “|” pipe operator creates a “future binding”, which is an object similar to a late binding (remember “&<symbol>?”), but having a value that may be used “in the future”. This “future”, in function calls, are the actual parameters. In fact, we may even create dynamic future bindings with the `lbind()` function (passing a second parameter), and/or store the future binding in a variable for later usage. For example:

```
alphaFuture = alpha | "The future value of alpha"
whatParams( beta | "Hello",
  lbind( "gamma", "Gamma value" ),
  alphaFuture )
```

We see the future binding value (which can be any expression) assigned to an item (alphaFuture) and then generated through the “lbind” function.

The Falcon Programming Language: a brief tutorial

Named and unnamed parameters can be mixed in the same call. In this case, unnamed parameters will fill the available parameters and after that all the parameters are assigned, the named parameters are taken and applied. So, the result of the following test:

```
whatParams( "will go in alpha",
            gamma | "Gamma value",
            "will go in beta" )
```

would be:

```
Alpha was: will go in alpha
Beta was : will go in beta
Gamma was: Gamma value
```

As you can guess, since future bindings are themselves Falcon items, they can be cached in array calls and mixed with normal parameters:

```
.[ whatParams
   beta|"in beta"
   gamma|"in gamma" ]()
```

Notice that lists in round and square parenthesis can be broken on multiple lines.

Falcon supports also *pass-by-reference* protocol. In other words, it is possible to let the called function modify the incoming value, returning something else in “output parameters”. As with many other things, this is done in Falcon via a special binding called *Item alias* or *reference*.

In the following code, typed directly in the interactive interpreter, we modify the `b` variable, and see what happens to `a`:

```
>>> a = "Value"
: Value
>>> b = $a // (1)
: Value
>>> b = "New value" // (2)
: New value
>>> > @ "We have a => \"$a\" and b => \"$b\""
We have a => "New value" and b => "New value"
```

In (1), the alias value to `a` is created and stored into `b`. Now, the assignment in (2) acts as if `a` was directly assigned; in fact we see that “`a`” is now changed, and both `a` and `b` can be used to change the same value. Along this principle:

```
>>> // (1)
>>> function mul2(x,y,result);result = x*y;end
>>> value = nil
: Nil
>>> mul2( 2, 4, $value )
>>> > value
8
```

Notice the “;” separating statements that should go on different lines in (1).

We already scratched variable parameter convention. It is possible to access directly the parameters passed to a function via a set of functions, the most important of which are `parameter()` and `paramCount()`.

The Falcon Programming Language: a brief tutorial

```
function varCall()
  for i in [0 : paramCount()]
    > i, ":", parameter( i )
  end
end

varCall( "one parameter" )
varCall( "one parameter", "two parameters" )
varCall( "one parameter", "two parameters", "three parameters" )
```

Static function data

Function local variables are normally reset each time the function is called but each function may have a “static” block; this block is executed only once, that is, the first time the function is called and all the local variables declared in the static block retain their values between calls. For example:

```
>>> function counter()
... static
...   > "Counter initialized!"
...   c = 0
... end
... return c++
... end
>>> > "First call: ", counter()
First call: Counter initialized!
0
>>> > "Second call: ", counter()
Second call: 1
>>> > "Third call: ", counter()
Third call: 2
```

The falcon VM has support for serialization and restore of function items; when serialized, the value of their static data are safely stored and re-created so it’s possible to preserve the state of the program across different sessions.

Object oriented programming

Enough of functions for now (we’ll see more in functional programming). Let’s talk now of that very funny thing that is the `object.member` notation that is so glamorous as to have conquered the vast majority of many programming language market areas.

Falcon allows both for class/type based object-oriented programming and prototype-oriented programming; that is, you can both create classes, derive from them, do interface-like inheritance, have private members, and the kind of things you usually expect in strongly typed object-oriented languages on one hand; and on the other, have loose objects created out of the blue, with shifting and changing members. Even more, we’ll see how tabular programming integrates and extends OOP into something quite new.

Basically, classes are a collection of properties, methods and attributes; once declared they become callable and calling them creates an instance:

```
class Alpha
  prop = 0
end

item = Alpha()
```

The Falcon Programming Language: a brief tutorial

```
item.prop = "Setting a property"
> item.prop
```

As with any Falcon item, classes are values and can be cached elsewhere and then instantiated by calling the cached item; continuing the above program we may see:

```
class Beta; pbeta = 1; end

the_class = random( Alpha, Beta )
inspect( the_class )

item = the_class()
```

We have a 50% chance to create an item of class Alpha or Beta.

See the result of an `inspect()` on the last item on a test run:

```
>>> inspect( item )
Object of class Alpha {
  prop => int(0)
}
>>>
```

A more complete class declaration is like the following:

```
class Name ( ... parameters ... ) \
  from ClassA( ...params... ), \
  ClassB( params ) ...
  ... properties ...
  ... init ...
  ... methods ...
  ... attributes ...
end
```

Actually, the distinction between properties and methods is not very important in Falcon, as callable items aren't treated specially. It is more a visual convention to have properties separated from methods. The following example is more complete:

```
class Vector( a, b, c )
  x = a ? a : 0
  y = b ? b : 0
  z = c ? c : 0

  init
    > @ "Vector initialized as $self.x, "
      "$self.y, $self.z"
  end

  function modulo()
    return ( self.x * self.x +
             self.y * self.y +
             self.z * self.z ) ** (1/2)
  end
end

v = Vector( 1, 2, 3 )
> v.modulo()
```

The Falcon Programming Language: a brief tutorial

Properties can be initialized through even complex expressions, using the passed parameters as initializers; the `a ? a : 0` parameter allows you to set a default value if a parameter is not given (`nil`). The `init` block receives the same parameters that are seen by the properties initialized and it is executed immediately after all the properties are set. The following function declaration creates a property containing a computational unit (callable); we may call this method, but the distinction between non-callable and callable data is very thin in Falcon. In fact, the vector class declaration is equivalent to:

```
class Vector( a, b, c )
  x = a ? a : 0
  y = b ? b : 0
  z = c ? c : 0

  modulo = function()
    return ( self.x * self.x +
             self.y * self.y +
             self.z * self.z ) ** (1/2)
  end

  init
    > @ "Vector initialized as $self.x, "
      "$self.y, $self.z"
  end
end
```

Or even:

```
class Vector( a, b, c )
  x = a ? a : 0
  y = b ? b : 0
  z = c ? c : 0
  modulo = nil

  init
    > @ "Vector initialized as $self.x, "
      "$self.y, $self.z"

    self.modulo = function()
      return ( self.x * self.x +
               self.y * self.y +
               self.z * self.z ) ** (1/2)
    end
  end
end
```

Other than instances, in Falcon it is possible to create classless objects, either directly or deriving them from one or more classes.

```
object AnObject
  prop = 100

  init
    > "The object has been created"
  end

  function inc(): self.prop++
  function dec(): self.prop--
end

AnObject.inc()
```

The Falcon Programming Language: a brief tutorial

```
> AnObject.prop
```

The main difference between stand-alone objects and normal instances is that their initialization is performed before the main code of a script is executed.

Falcon supports multiple inheritance with explicit order overloading:

```
class Base1
  prop = 100
  function common(): > "From Base1"
  function b1(): > "Personal b1"
end

class Base2
  prop = 200
  function common(): > "From base 2"
  function b2(): > "Personal b2"
end

class Derived from Base1, Base2
end

instance = Derived()
> instance.prop
instance.common()
instance.b1()
instance.b2()
```

As seen, the class declared last in the “from” clause (in our case, Base2) takes precedence over the others in forming the final instances. It is possible to access an arbitrary base class element by explicitly naming it; continuing the previous example:

```
instance.Base1.common()
```

One very important concept in Falcon OOP model is the *method persistence*. Look at this example typed at the interactive interpreter:

```
>>> array = [1,2,3]
: Array
>>> method = array.len
: <?>
>>> array += 4
: Array
>>> > "Current array len: ", method()
Current array len: 4
```

The method taken from the array instance (`len` is a method available for any Falcon item) continues to refer to the given item also if the item changes.

That’s enough for now. I won’t go into more technical aspects of the class and object definitions or the static members, static calls and static function blocks, or the initialization and inter-module class resolution sequence. Just notice that Falcon classes are seen internally as types and it is possible to determine the type of an object and its inheritance structure through language facilities. For example, `select` is a special statement that switches over the type of a variable; as such, it is possible to write something like the following:

```
select item
  case StringType
```

The Falcon Programming Language: a brief tutorial

```
    ...
case Derived
    ...
case Base2
    ...
case Base1
    ...
end
```

One quite useful language feature is the “provides” operator which checks for a property being actively offered by an instance. For example:

```
if instance provides method
    instance.method(...)
    ...
end
```

Class instances and objects can be given one or more attributes. Attributes are binary qualities that can be checked for quite effectively and can be assigned to every class and object instance. Also, it is possible to iterate through all the items having the same attribute at a certain moment and record the list of items with language functions. Look at the next example (not suitable for the interactive compiler):

```
attributes
    isReady
end

class CanBeReady( id )
    id = id                                //(1)

    init
                                           //(2)
        if random(true, false): give isReady to self
        end

    function hail()
        > "Hello from CanBeReady ", self.id
    end
end

object BornReady
    function hail()
        > "Hello all"
    end

    has isReady                            //(3)
end

// create some instances.
insts = []
for i in [0:10]
    inst += CanBeRead( i )
end

// ok... who's ready?
for item in isReady                        //(4)
    if item provides hail
        item.hail()
    end
end
```

Notice that properties can be named after the parameters of the class initializer as in (1). In (2) I use the “give” statement that can assign or remove one or more attributes to/from one or more objects. In (3), I declare the object has having the `isReady` attribute from the start. The same “has” clause can be specified also in classes, so as to make their instances have an attribute since their creation. In (4) I use the for/in loop to traverse all the items having an attribute. There are various other ways to scan the “attributed set”, or the set of instances having a certain attributes; in this example, it is possible to generate an iterator (Java-like list traverse facility object), or call the `having(...)` function that returns an array containing all the items in an attributed set. The `has` and `hasnt` operators can be used to check for an instance having a certain attribute.

Prototype-oriented programming

Prototype-oriented programming is “light object oriented programming” where emphasis is set upon the exposition of commonly agreed interfaces rather than on class and derivation.

There are two devices in Falcon that work like that: blessed dictionaries and array bindings. Falcon dictionaries are collections of pairs of unique keys and values. Keys can be any valid Falcon item, possibly providing an internal ordering (a compare method), or, in case this is not provided, ordered on their unique memory representation. Try this sample to get an idea:

```
date = CurrentTime()

dict = [ 1 => "Data for key 1",
        "alpha" => "Data for key alpha",
        date => "Data for a timestamp" ]
inspect( dict )
```

They can be accessed and modified like arrays:

```
> dict[ "alpha" ]
dict[ "alpha" ] = "Changed"
> dict[ "alpha" ]
```

For the matter at hand, it is simply possible to create objects by creating methods which refer to `self` and access the inner data both as dictionary items and as properties. It is just necessary to “bless” the dictionary; in this way, we tell Falcon that we want the dictionary to use its own string keys as properties:

```
instance = bless( [
    "prop" => 0,
    "inc" => function(); self.prop++; end,
    "dec" => function(); self.prop--; end ] )

instance.inc()
> instance.prop
```

An interesting property of Prototype OOP is that it is possible to change the definition of the instances after their creation; continuing the above example:

```
instance[ "reset" ] =function();self.prop = 0;end // (1)
instance.reset()
> instance.prop
```

The Falcon Programming Language: a brief tutorial

In (1), a new entry in the dictionary is created and it is set to a method resetting a property. As in full blown OOP, Prototype OOP is subject to method persistence, but method persistence is not mandatory. In simpler words, it is possible to extract a complete method, recording the original instance from which the method was taken, or just take the data as if the item was a simple dictionary, breaking the method-instance unity. Still continuing the above example:

```
method = instance.inc
method(); method(); method()           // (1)
> instance.prop
```

In (1) we're using the method unity, but...

```
func = instance[ "inc" ]
func()                                  // (2)
```

doing this would raise an error in (2), as `self` becomes unassigned. This means we can take methods stored in blessed dictionaries and assign them to other objects or blessed dictionaries, and the `self` item stored inside them will refer to the target item. For example, instead of (2) we can do:

```
other = bless([ "prop" => 0, "inc" => func ])
other.inc()
> other.prop
```

If we used `instance.inc` to get the `inc` method in `instance`, putting it in any object would have cut the method unity, and...

```
method = instance.inc
other = bless([ "prop" => 0, "inc" => method ])
other.inc()
> instance.prop                       // (3)
```

In (3) we see that `self` still refers to `instance`, even if called from `other`.

Notice that the `self` object is resolved as the original item in which the method refers to the actual dictionary where the method is stored. So, it is possible to modify dynamically the dictionary from within a method; see this example:

```
instance = bless( [
  "counter" => 0,
  "addMethod" => function();
    x = ++self.counter;
                                // (1)
    self[ "newMethod" + x ] = function();
      > "I am new method " + x;
    end;
end ])

instance.addMethod()
instance.addMethod()
instance.addMethod()

instance.newMethod2()                // (2)
```

The `"newMethod" + x` expression in (1) will cause a new entry to be generated by concatenating the string `"newMethod"` and the string literal value of `"x"` (1, 2, 3 etc). Notice that the `;` at the end of the statements inside the dictionary declaration is necessary, as the parser suspends the recognition of end of lines

The Falcon Programming Language: a brief tutorial

when it enters a set of parenthesis or square brackets.

The second mean of prototype oriented programming merges with the array bindings we have seen previously. In fact, other than data, array bindings may also contain functions and functions referring to `self` stored as array bindings become methods. See the following “doubler” example:

```
doubler = .[ map &func ]
doubler.func = lambda x => x * 2
inspect( doubler( [1,2,3,4] ) )           //(3)
```

The “map” function uses the first parameter it receives, applying it to each element of the second parameter. So, if the parameter is a binding it can be configured later on via assignment to the given binding. The effect of (3) will be that of doubling each element in the given array and showing the result. However, array bindings are not limited to being used internally via the `&` operator. They can work very similarly to blessed dictionaries:

```
instance = []
instance.prop = 0
instance.inc = function(); self.prop++; end

instance.inc()
> instance.prop
```

We’ll see how this is integrated in functional programming and tabular programming in the following paragraphs.

One last note: to create multiple instances of a prototype object we just need the language-wide “clone” method. Continuing the previous example:

```
copy = instance.clone()
copy.inc()
> @ "Original value: $instance.prop; Modified value: $copy.prop"
```

Functional programming

We can just touch upon this deep topic here. This programming paradigm would require a specific introduction course. However, we have seen some functional aspects in Falcon while describing other characteristics.

The Falcon functional programming model is based on evaluation of sequences, which can contain normal function calls or special functions called *Eta*, which redefine internally the functional evaluation process.

For example, the functional `if` (`iff`) resolves one branch or another depending on the evaluation of a comparand. Some useful functions for functional programming can be found in the “funcext” Falcon standard module. The following example uses `gt` (greater than) from the `funcext` module:

```
load funcext

iff ( .[gt .[readNumber] 10 ],           //(1)
     .[printl "The nubmer is > 10" ],
     .[printl "The number is <= 10" ] )

function readNumber()
  while true
```

The Falcon Programming Language: a brief tutorial

```
>> "Please enter a number: "  
try                               //(2)  
  return int( input() )  
end  
end  
end
```

The code in (2) is a “neutral” try, discarding any error coming from the code in its block; in fact, `int()` would raise an error if the user doesn’t input a number, in which case we’d just ignore the error and loop again. The `gt` function in (1), from the `funcext` module, performs a “greater than” check on the two elements, and is equivalent to:

```
gt = lambda x, y => x > y
```

Notice in (1) the fact that `readNumber` function is itself in stored in a separate sequence; in this way, the evaluation engine working on the comparison code sees that the item must be reduced, and does so by resolving (calling) the function `readNumber`.

The simplest and most generic interface to the functional engine in Falcon is the `eval()` function and the evaluation operator `^*`. Supposing we store the sequence for a later evaluation, it is possible to configure them via late bindings. Continuing the previous example:

```
seq = .[iff .[gt .[readNumber] &limit ]  
  .[printl "The number is > " &limit ]  
  .[printl "The number is <= " &limit ]  
  ]  
  
seq.limit = 5  
eval( seq )                               //(1)
```

Or instead of the `eval` function in (1):

```
^* seq
```

The `^*` `eval` operator is quite useful whenever you are unsure about the nature of a property or of a variable. If the variable is a value, then that value is returned; if it’s a function, a method, a functional sequence or any callable item, it is called and its return value is set as the result of the expression.

Falcon provides a rich and ever growing set of functional operators and Eta functions. The following code shows the functional version of a `for/in` loop counting only pair numbers.

```
^* .[times ,[0:11] &n .[  
  .[ (lambda x => x % 2 != 0 ? oob(1):0) &n ]  
  .[ printl "Pair number: " &n ]  
  ]]
```

The `times` function calls repeatedly the functions listed in its third parameter, assigning the binding passed as second parameter a value determined by the first parameter. In this case, we used a range. To avoid confusion with a ranged access to the “times” symbol, we used an explicit comma to separate it. `Times` then assigns 1, 2, 3 and so on to the `&n` binding, calling each time the functions in the following sequence. In our case, the first function may return either 0 or an `Out of band` 1.

Out of band items are items with special meaning in functional sequences or in your program. I can’t delve into details about their usage patterns here but the out-of-band flag can travel with an item, be it stored as an

The Falcon Programming Language: a brief tutorial

array or dictionary value, as a property, passed as parameter or returned from a function, to signal the algorithms involved in the process about “special meanings”. For example, many functions may return legally any Falcon item, including nil, but they may wish to inform the caller about an “invalid” data or a return value to be ignored without being forced to raise an error. In these cases, the processor function may return an out of band nil item to tell the caller to ignore the value.

The “times” functions, and all the other functional loops provided by the standard library of Falcon, interpret an out of band “1” as a request to restart the loop advancing the loop count variable. In other words, `oob(1)` is a sort of “functional continue”. So, when the `&n` binding parameter passed to the lambda as “x” is found to be impaired, the lambda expression returns an out of band 1, asking “times” to skip the rest of the sequence.

Similarly, an `oob(0)` is considered a functional break. The next example exits as a pair number > 5 is found:

```
^* .[times , [0:11] &n .[
    .[ (lambda x => x % 2 != 0 ? oob(1):0) &n ]
    .[ printl "Pair number: " &n ]
    .[ (lambda x => x > 5 ? ^+ 0 : 0) &n ] // (1)
  ]]
```

Notice the “^+” operator in (1); this operator works exactly as the `oob()` function, making the following item an out of band item.

Tabular programming

Tables are simple but powerful means to represent facts and to organize things. Columns are properties, and row instances; a cell can contain any Falcon item, including OOP instances, classes, functions and even other tables. So, cells can be data but also algorithms or even whole programs. Tables are also an excellent way to select behaviors between a finite set of choices, or to mix behaviors and merge them, providing ready-made fuzzy logic engines.

The heart of tabular programmings is the Table class and Falcon arrays, which performs as table “rows”. Arrays used as table rows know the table which they come from, and their elements can be accessed by name through the dot accessor, or still be referred by numbers (column indexes).

For example, let’s create an employee table which includes a configurable payment method:

```
function normal_salary( work_hours )
  return work_hours * self.hourly_pay // (1)
end

function extra_salary( work_hours )
  if work_hours <= self.daily_hours
    return work_hours * self.hourly_pay
  else
    return self.daily_hours * \
      self.hourly_pay + \
      (work_hours - self.daily_hours) * \
      self.extra_pay
  end
end

Employees = Table( // (2)
```

The Falcon Programming Language: a brief tutorial

```
.[ 'name' 'hourly_pay' 'extra_pay'
   'daily_hours' 'payment' ],
.[ 'ed'    20          24
   8          normal_salary ],
.[ 'frank' 18          24
   6          extra_salary ] )

// (3)
> "Ed's pay for 9 hours: " + \
  Employees.find( 'name', 'ed' ).payment(9)
> "Frank's pay for 9 hours: " + \
  Employees.find( 'name', 'frank' ).payment(9)
```

You can see that table rows can be accessed as objects, their properties being the column names of the table in which they are stored. The function in (1) is set as part of a row, and it retrieves the value of the “hourly_pay” column of the active entity. The call in (2) creates the table. In (3), the “find” method gets the table row where the entity with the given name is found, and then it calls the “payment” property, resolving in one of the two calculation functions. It is also possible for a row to refer the owning table via “self.table()”.

Tables can be widely manipulated; rows and column can be inserted and removed, and a set of table specific algorithms is provided. As the model has been just recently introduced, the algorithms are currently limited to essential searches but it is also possible to perform a bidding or choice depending on the best row given a choice algorithm. Tables make a great base from which to pick different partial solutions and mix them. Simple, ready-made fuzzy logic engines can be built by selecting a row depending on the fitness of some of its elements. Then, several functions residing in the chosen row can be applied and a weighted mean value can be then determined. The next versions of the engine will have a strong support for this table-wide operations.

Rows in tables are still Falcon arrays. Continuing the above example, we can simply...

```
>> "Data regarding ed: "
for item in Employees.get(0)
  >> item
  formiddle: >> ", "
  forlast: > "."
end
```

And, as normal arrays, it is possible to give them bindings that are not part of the table structure. For example:

```
ed = Employees.find( 'name', 'ed' )
frank = Employees.find( 'name', 'frank' )

ed.wife = "mary"
frank.fianc? "jenny"
```

But mangling the array indexes is immediately reflected in the corresponding table property, and the other way around:

```
ed.daily_hours = 9
> "New daily hours for ed: ", ed[3]

frank[3] = 8
> "New daily hours for frank: ", ed.daily_hours
```

Tables can be ordered into pages, to store dataset that are suitable under some conditions and can be switched when other conditions occur.

Message oriented programming

As tabular programming, this model has been recently introduced and is being extended and reviewed.

The idea beyond message-oriented programming is that of creating a set of agents which can compete or co-operate on messages, which can carry any Falcon data. Instead of calling directly a target method in a certain object, a sender generates a message that can be received by one or more receivers. The data traveling along with the message is not read-only; this means that participants to the broadcast process may contribute in creating a common solution, rather than immediately providing one.

Attributes are currently the means by which subscription to messages are performed. By having an attribute, an object declares it is willing to receive broadcasts on that attribute. Broadcasts are then sent to a processing method in the object, which has the same name of the attribute:

```
attributes
  listening
end

class Agent( id )
  id = id
  init
    > "I am agent ", id
  end

  function listening( p1, p2, p3 )           //(1)
    > @"Processing message $p1 $p2"
    if p3.type() == ArrayType
      p3 += self.id                         //(2)
    end

  end

  has listening                             //(3)
end
```

The method, having the same name as the attribute in (1), will be called back when a message on the attribute (3) is broadcast. The code in (2) just shows a sample of co-operative creation of a shared data. Each agent participating in the broadcast just adds its ID to a list being formed.

A sample usage of this class may be:

```
alpha = Agent( "alpha" )
gamma = Agent( "gamma" )
delta = Agent( "delta" )

cooperate = []
broadcast( listening, "Hello", random(100),
          cooperate ) // (1)

// who did cooperate?
for agent in cooperate
  > "Agent ", agent, " participated."
end
```

The function broadcast (1) passes the parameters that follow to all the “listening” methods registered by giving the attribute with the same name to an instance. As the third parameter is an array, it will be filled with

The Falcon Programming Language: a brief tutorial

the ID of the items receiving the message. Of course, it may be any Falcon item, including a method, a whole functional sequence, a class or an instance and so on.

There are several mechanisms associated with the broadcast function. Let me just enumerate them:

- Returning “true” from a receiving method makes it possible to stop the message being sent to further receivers. This allows to consume the message in a non-cooperative (exclusive) message management mode.
- It is possible to send messages to “priority queues”. The first parameter of the broadcast function is a list of attributes; then the message broadcast is repeated for each attribute until a method responding to the message returns true.
- It is possible to perform asynchronous broadcasts by using the launch keyword, which starts a co-routine calling the required function. For example, `launch broadcast(listening, "Hello", random(100), cooperate)` would immediately return preparing the VM to perform the broadcast via a co-routine.
- To subscribe or unsubscribe a broadcast, it is possible to either add or remove the attribute from an instance, or eventually nil the receiving method.

One of the things that is already implemented in this model is the event marshaling system. When an application needs to send articulated messages to a relatively constant set, instead of listening on different attributes, it is possible to use the event Marshalling mechanism, which also provides facilities to raise errors in case of unhandled events. Events are structured as an array whose first element is a string, which will be the event name. The other elements of the event array will be received as parameters of the event handlers.

For example, suppose that we want to send two “orders”, activate and deactivate, to all the listeners. Continuing the previous example:

```
class ActiveAgent( id ) from Agent( id )      // (1)
  listening = .[marshalCBR "on'"]           // (2)

  function onActivate( p )
    > self.id, " activated with ", p
  end

  function onDeactivate()
    > self.id, " deactivated."
  end
end

aa1 = ActiveAgent( "active one" )
aa2 = ActiveAgent( "active two" )

broadcast( listening, [ "activate", random(10) ] )
broadcast( listening, [ "deactivate" ] )
```

The inheritance declaration in (1) spares us some detail, as adding the listening attribute. The code in (2) overloads the listening method with the `marshalCBR` function. The first parameter, `on'` just tells the system to marshal the incoming event to the method starting with `on` and the first letter of the event capitalized. When the listening method is called with the event as an extra parameter, the event is decomposed and the proper method is called.

We're working at a more complete message-oriented model which should make it possible to write programs that don't require calling or prior knowledge of any non-system module.

Other stuff I couldn't squeeze in

Even if what has been discussed to date is detailed, Falcon has still more important features that cannot be covered in this article. The most important are:

- FTD - Falcon template documents (php-like template-based scripting).
- Exception management (try/catch/raise statements).
- Co-routines (we have seen the `launch` command).
- Native support for internationalization.
- Multi-platform native zero contention multi-threading.
- Module loading, relative naming and active namespaces.
- On-the-fly compilations.
- Plugin-ability (dynamically loadable modules).
- Meta-macro compiler.

The engine offers an unmatched level of facility for embedding applications, which can extend the virtual machine, create streams that can be fed into the virtual machine to manage directly its I/O, co-operative time slice management, limits control, periodic callback control, direct C++ structure reflection into language classes and more. Even more support is granted by a `Service` class hierarchy which shares the code offered to Falcon scripts with embedded applications or simply with foreign applications; that just need to use the Falcon dynamic library loader to access a set of multi-platform and virtualized services that Falcon modules offer to scripts.

There are many things that Falcon has to offer to the community, but we need the community to support Falcon. The project is young but we have already a wide and flexible codebase. If you want to get your hands dirty in a project like this, bring your own ideas and help us to make them a reality. We will welcome you.

Giancarlo Niccolai (The Falcon Programming Language)

Dennis M. Clarke (Blastwave.org Inc.)

Biography

[Giancarlo Niccolai \(/user/60376](/user/60376) title="View user profile.): Graduated in IT and Majored in Economics, he currently works in the Financial IT area as a professional application designer. He is in charge of planning and actively participating in the development of software meant to distribute real time market data and integrating with financial trading facilities. He developed the Falcon Programming Language to support real-time analysis of data streams in its daily job, and as a general purpose development tool for IT professionals. He's specialized in business strategy and decision support systems making processes, where he applied AI and A-Life techniques.

[Dennis Clarke \(/user/61704](/user/61704) title="View user profile.): CEO of Blastwave.org Inc and director of technical software operations for the Blastwave(tm) Software Stack. The primary objective of Blastwave(tm) is to provide free access to open source software for current Solaris users. Blastwave currently delivers 1.2M software packages from a tree of 4000+ software packages to users of Solaris 8, Solaris 9 and Solaris 10 as well as users of OpenSolaris(tm). Blastwave is an avid supporter of community activities. Blastwave provides open source software and hardware facilities to community members that want to port software, experiment with Solaris or OpenSolaris or even create a distro. Software support options will be made commercially available in Quarter 1 2009.

Copyright information

This article is made available under the "Attribution-NonCommercial" Creative Commons License 3.0 available from <http://creativecommons.org/licenses/by-nc/3.0/>.

Source URL:

http://www.freesoftwaremagazine.com/articles/falcon_programming_language_brief_tutorial
