The Falcon Programming Language

Quick start

Niccolai Giancarlo

– 0.8.14 –

Table of contents

Foreword....2
In the beginning....3
    Calling the interpreter....3
    Functions and branches....3
    Lists and iterations....4
    Functional loops....6
    Function expressions and nesting....8
    Calling conventions....9
Classes and objects....12
    Method persistence....13
    Inheritance....14
    Singleton objects....16
    Blessed dictionaries....16
    Array bindings....17
Tabular programming....18
    Table are classes....18
    Default Values....19
    Table operations....20
    Table pages....21
    Table-wide operations....24

Foreword

This document is a quick and dirty introduction to the basic Falcon concepts. It's intended for those who know basic programming concepts. The target reader of this document knows what's a branch, an operator, a variable, a loop and so on.

As Falcon introduces some new concepts and bends some old concepts into something quite different, we're not expecting the reader to know everything about programming. Although we expect you not to be surprised by an if/else branch, a while loop, or what a return is supposed to do. More or less, you don't need to know anything else; and even if you know more, you may want to unlearn and relearn what you know once you understand how powerful an old concept can become if is gently pushed a bit forward, or mixed with something else.

Also, we're not starting exactly from the basics. The basics (for example: making sums, math expressions, accessing strings, and well... other basic things) are important for sure, but they're terribly boring and nowadays also terribly standard. Or to be more precise, Falcon does the basics in a pretty standard/old fashioned way. In our opinion, trying to be fancy with the basics may look cool on a quick start guide and can even save a few keystrokes here and there, but it is not what makes programming easier or more fun. Being fancy will however allow for some early, surprising effects.

In the beginning...

Falcon is a powerful programming language, a scripting engine and an interpreter that can be run from the command line of most modern operating systems. It provides users with six programming paradigms (procedural, object oriented, prototype oriented, functional, tabular and message oriented), and comes with an ever growing set of modules which extends the functionality of the base language.

While maintaining a relatively common set of basic concepts that can be found in modern typeless scripting languages, Falcon introduces some new concepts that allow a mix of different programming paradigms in an unique blend. Yet, each one of the available programming styles can be used separately. You can pick the style you feel most comfortable with, or the one more appropriate to solve the situation at hand.

Calling the interpreter

First, we'll invoke Falcon from the command line as an interactive interpreter. From the command line, type

```
$ falcon -i
```

A disclaimer and an arrow shaped prompt will inform you that Falcon is waiting for expressions to evaluate. For example,

```
>>> 1+1
: 2
```

Falcon will show the result of the last evaluation prefixed with "`:`". Let's get some confidence with the basic I/O means that we'll use in this tutorial.

The `printl` function prints a list of comma separated items; as does the "fast print statement", which can be invoked by writing a single "greater than" symbol in front of a line. The inspect function gives additional information about the contents of a variable, describing its type and navigating through components of complex items.

So, a simple "Hello world" can be written as...

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

Notice that inspect has shown the string surrounded by quotes to indicate its type.

Functions and branches

Other than using predefined functions, we can also define new functions in our programs. The following function determines the age of the caller and hails correctly.

```
>>> function hail( name, age )
```

As soon as we enter the function declaration statements, which tells Falcon we want to declare a function that should receive two parameters, the prompt turns into a "`...`" indicator. This indicates that a lexical context has been opened, and that no evaluation will be performed till the matching context closing statement (in our case, end) is found.

```
...    if age < 0
...       > "Sorry, you're way too young"
```

The `if` statement tells Falcon that the following statements are to be executed only if the specified condition is matched.

```
...   elif age <= 12
...      > "Hello ", name
```

The `elif` statement indicates that following statements are evaluated only if the condition is valid and previous `if` or `elif` conditions are not matched. So, the above print statement is to be executed only when age is greater than 0 but less or equal to 12.

```
...   elif age <= 21
...      > "Yo ", name, "!"
...   else
...      > "Good morning, Mr. ", name
...   end
```

Finally, the `else` statement requires evaluation of the following statements only if none of the `if` and `elif` conditions were matched. The if-elif-else block is closed by an `end` statement. Both `elif` and `else` are optional, so an if-elif-else branch may just be formed by an `if`, related statements and `end`.

We still need to close the function:

```
... end
```

Now the prompt returns into the usual “>>>” where we can try to call our new function by passing some values to it.

```
>>> hail( "Jane", 4 )
Hello Jane
>>> hail( "Eric", 15 )
Yo Eric!
>>> hail( "Sanders", 40 )
Good morning, Mr. Sanders
```

Lists and iterations

Falcon has several ways to iterate through a list of elements. First, let's create the basic type of list, the array, and store it in a variable:

```
>>> data = ["john","mary","jane","edward"]
: Array
```

The string ": Array" is printed to inform us that data was parsed and stored as an array. We can see what's inside it using inspect:

```
>>> inspect(data)
Array[4]{
   "john"
   "mary"
   "jane"
   "edward"
}
```

Array elements can be accessed by specifying a zero-based index between brackets; for example to find the third element in the array...

```
>>> data[2]
: jane
```

The index “2” retrieves the element n.2, which is the third element (as the first is numbered zero). Knowing this, we can see what's in the array by using an integer variable that is incremented at each loop:

```
>>> i=0
: 0
>>> while i < data.len()
...   > "Item ", i, ": ", data[i]
...   i=i+1
... end
Item 0: john
Item 1: mary
Item 2: jane
Item 3: edward
```

Notice that after inserting the "`while`" command, the prompt changed into a "`...`". This is because falcon recognizes that `while` is a *block statement*, which repeats a list of statements comprised between itself and the matching "`end`" statement.

The while loop iterates until the condition specified in the `while` line becomes false. In this case, it loops until i becomes greater than or equal to the size of the data array. At each loop, the i variable and the corresponding element in the data array are fast printed. Then, the variable is incremented by assigning to itself its current value plus one.

The same effect can be achieved by rewriting the above while loop in a more compact form:

```
>>> while i < data.len(): > @ "Item $i: ", data[i++]
```

The "`:`" symbol indicates that the block instruction will actually contain only one statement, which will be written on the same line. This is a shortcut for writing blocks on one line. The "`@`" symbol is the *string expansion operator*. Strings and variables following that operator are parsed in search of "`$`" followed by a variable name. The current value of the variable is then substituted into the string and returned. The "`++`" operator works exactly as the homonym operator found in the C language, incrementing the variable near to it.

We'll see other constructs for language components now, as there is a loop which may be more useful to traverse sequences: the for/in loop.

```
>>> for name in data
...    > @"Hello $(name)!"
... end
Hello john!
Hello mary!
Hello jane!
Hello edward!
```

The variable name, which is specified inside the block between the `for` statement and the `end` keyword assumes the value of each element in the data vector in turn. Notice also the parenthesis near the "`$`", which can be used to disambiguate variable names in string expansions.

The for/in loop can contain three special blocks: `forfirst`, `formiddle`, and `forlast` which are executed respectively before the first element, after every element except the last one, and after the last element. For example, a well formatted list can be printed with:

```
>>> for name in data
...     forfirst: >> "Hello to "
...     formiddle: >> name, ", "
...     forlast: > "and ", name, "."
... end
Hello to john, mary, jane, and edward.
```

Notice that the ">>" operator was used to write text without a newline after it. Being block instructions, `forfirst`, `formiddle` and `forlast` can be shortened with the colon "`:`" separator. Also, it is possible to write instructions outside the special blocks to have them executed every time. We can achieve a similar effect with this code:

```
>>> for name in data
...     forfirst: >> "Hello to "
...     >> name
...     formiddle: >> ", "
...     forlast: > "."
... end
Hello to john, mary, jane, edward.
```

Functional loops

Falcon provides a wide range of functional operations to perform loops and iterate over lists. The functions `dolist` and `times` allow each element of the list to be worked on via functional constructs. Also mapping and filtering operations are provided, but we will see them in a later section.

For now, let's just introduce the simplest functional loop possible:

```
>>> dolist( printl, data )
john
mary
jane
edward
```

The `dolist` function calls iteratively the function in the first parameter; passing each element taken from the list specified in the second parameter to the function one each time.

Now, we have to take a step back and see how functions can be made more "interesting", so that they can be used in ways that are more creative than just receiving a parameter at a time. For this reason, let's examine another way to call functions:

```
>>> printl( "Hello this way" )
Hello this way
>>> [printl, "Hello another way"]()
Hello another way
```

Yes, in Falcon you can call arrays (and a whole lot of other things), and you can even call them in pretty creative ways, like in the following example:

```
>>> f = .[ printl "%%%= " ]
: Array
>>> f( "A value" )
%%%= A value
```

The first instruction is a *comma-less* array declaration. Declaring an array with the "`.[`" (dot-square) symbol instead of the "`[`" (square parenthesis) specifies that when the array items are written, they will be separated by just white space instead of commas.

In other words, f is just an array, in fact...

```
>>> inspect(f)
Array[2]{
   Ext. Function printl
   "%%%= "
}
```

Calling it will actually call the function on top of the array, which will receive the other array elements and the other parameters passed dynamically in the call as its own parameters. So,

`.[ func p1 p2 ... pk ]( pk+1, ..., pn ) == func( p1, p2, ..., pn )`

Knowing this, we can make our little loop a bit more interesting:

```
>>> dolist( .[ printl "Item: "], data )
Item: john
Item: mary
Item: jane
Item: edward
```

This is fun, but what if we want our item to not be the last element passed to the processing function?

In that case, we should declare a temporary function that takes the parameter which is given it by `dolist` and then places it in the right position. The "`lambda`" keyword declares a lambda (or "parametric") expression.

```
>>> dolist( lambda a => printl( "Item: ", a, "." ), data)
Item: john.
```

```
Item: mary.
Item: jane.
Item: edward.
```

The parameter (or list of parameters) of a lambda expression are indicated after its declaration, while the expression itself is to be written after an arrow "=>" symbol. Lambda expressions are values themselves, and they can be recorded in variables for later usage and calls, as callable arrays:

```
>>> l = lambda a => printl( "Some value: ", a )
: Function _lambda#_id_2
>>> l( 100 )
Some value: 100
>>> dolist( l, data )
Some value: john
Some value: mary
Some value: jane
Some value: edward
```

In case the position of the current item in the list being processed needs to be known, the `times` function comes into help. The `times` function evaluates in turn a sequence of callable elements, passing them a value which changes at each loop. There are many ways in which this value can be given to the called elements, but what we want to show now is a type of item called *late binding*.

```
>>> l = .[ times data.len() &i .[ .[ printl "Item: " &i "."]]]
: Array
>>> l()
Item: 0.
Item: 1.
Item: 2.
Item: 3.
: 4
```

Late binding allows us to associate a variable with a value during array call and functional evaluation. The `times` function calls each callable item in the list a determined number of times (in our case, the length of the data array), taking the late binding "`&i`" and changing its value each time. Finally, it returns the last value that was assigned to the loop variable, the one that made the loop to terminate.

An interesting aspect of late bindings is that they can be inspected, and even modified, at a later time:

```
>>> l.i
: 4
>>> l.i = 5
: 4
>>> l.i
: 5
```

The "`&i`" late binding in the `l` array can be accessed via the dot operator, and it can be also modified (notice that while being modified, the value of the expression is the previous value that was formerly held in the binding).

So, can we get the elements in our data array via late bindings?

This is a bit more complicated. In fact, the following sequence won't work:

```
>>> l = .[ times data.len() &i .[ .[ printl "Item: " data[&i] "."]]]
AccessError RT0047 at falcon.intcomp.__main__:1(PC:56): Access array out of bounds (LDV)
```

In fact, array access (the `data[&i]` expression) is a live expression, which is evaluated while creating the `l` array; so the current value of `&i` is taken. To access data in an array during a functional evaluation, it must be accessed functionally. But there is another small complication:

```
>>> l = .[ times data.len() &i .[ .[ printl "Item: " .[lambda n => data[n] &i] "."]]]
: Array
>>> l()
Item: Array.
Item: Array.
Item: Array.
Item: Array.
```

```
: 4
```

Remember when I said that times “calls” each element of the given array? That's exactly what it does. The elements are called as they are, untranslated. So, `printl` sees just an array as the second parameter. To tell Falcon that we also want that array to be considered as a functional sequence to be evaluated, we must be explicit and ask for functional evaluation with the `eval` construct:

```
>>> l = .[ times data.len() &i .[ .[ eval .[ printl "Item: " .[lambda n => data[n] &i] "."]]]]
: Array
>>> l()
Item: john.
Item: mary.
Item: jane.
Item: edward.
: 4
```

The `eval` function evaluates functionally the sequences it receives as parameters. Functional evaluation means to recursively search for callable arrays and then call them, inner to outer, substituting their return value in the final call of their parent element (this is a bit of an informal definition, but we're in the middle of a tutorial, aren't we?). As this operation is expensive, the `times` function doesn't automatically perform it.

I know what you're thinking. “Does this all have to be so complicated?”

No, not really. I have just shown the “hard way”. We can make it simpler, and also see a very important feature of Falcon: module loading.

The modules known as “feathers” are considered fundamental for Falcon. Among them is a a module called `funcext` (functional extensions) which provides functions which are tailored for functional evaluations. One of the provided functions is the “`at`” function which returns the nth element of an array.

```
>>> load funcext
>>> at( data, 2 )
: jane
```

Also, we may use the `xtimes` function that unlike `times`, forces evaluation of each element in the loop.

```
>>> l = .[xtimes data.len() &i .[.[printl "Item: " .[at data &i] "." ]]]
: Array
>>> l()
Item: john.
Item: mary.
Item: jane.
Item: edward.
```

Function expressions and nesting

The function declaration is a top level statement; functions can be declared as we have seen only at a top level, that is, it is not possible to declare a local function inside another function. Trying that will raise an error:

```
>>> function test()
...     function test1()
SyntaxError C00136 at 40: Functions must be declared at toplevel
... end
>>>
```

But there is another way to declare functions which allow nesting: nameless functions may be used as expressions, in a similar way to lambda expressions:

```
>>> > "abs(-10)= ", (function(value); return (value>=0 ? value: -value); end)(-10)
abs(-10)= 10
```

Using the `function` keyword immediately followed by the parameter declaration opens a “nameless function”. The return statement returns a value determined by the “`?:`” fast if expression. This expression is similar to the corresponding C expression and returns the value before the “`:`” symbol if the expression before the “`?`” symbol evaluates to true, while it returns the value past the "`:`" if the expression is false. We have packed everything on a line using the separator "`;`", which works exactly as if an end-of-line was entered, because it is easier to write in the interactive interpreter. A real program would probably read:

```
(function(value)
   return value >= 0 ? value : - value
end)(-10)
```

Now, what exactly happened?

The fact is that nameless functions, once closed by their end, become valid expressions. As they represent functions, they are also callable. In the example, the nameless function was called by applying the `(-10)` call right after the declaration of our nameless function. Of course, as they are expressions, nameless functions can also be assigned to any variable, or used where we used lambda expressions in the previous examples. So:

```
>>> xbs = function(value); return value >= 0 ? value : -value; end
: Function _lambda#_id_2
>>> xbs( -10 )
: 10
```

Functions declared this way, as well as lambdas, perform a "closure". This means that local variables and parameters declared in the parent scope are "closed", frozen as they are and then used in the same state as when the function was called. For example:

```
>>> function makeMultiplier( amount )
...    return (function( value ); return amount * value; end)
... end
>>> m2 = makeMultiplier( 2 )
: Array
>>> m4 = makeMultiplier( 4 )
: Array
>>> m2(10)
: 20
>>> m4(10)
: 40
```

What's going on?

First, we've declared a function that has a parameter "amount"; that parameter is local to the outer function, so when we name it in the inner function, its value is recorded and fed in the return value. Falcon then tells us that the value is an array; in fact:

```
>>> inspect(m4)
Array[2]{
   Function _lambda#_id_2
   int(4)
}
```

The Falcon compiler has closed the desired value in this callable array, which can then be called the way we did or the array may be inspected and changed at a later time.

Calling conventions

Falcon adopts a relaxed calling convention; this means that the parameters you put in the call do not need to strictly match the parameters declared in the function declaration. If the caller provides more parameters than required, the extra ones will simply be ignored (but they can be retrieved nevertheless using language functions). If fewer parameters are provided than expected, the ones not given will be filled with nil.

This example demonstrates the concept:

```
>>> function testPars( one, two )
...    > @ "one: $one; two: $two"
... end
>>> testPars(1,2)
one: 1; two: 2
```

```
>>> testPars(1)
one: 1; two: Nil
>>> testPars(1,2,3)
one: 1; two: 2
```

Notice the "`@`" string expansion operator used to print the parameters.

Falcon also supports the named parameter call convention; in other words, you can directly name what parameter(s) you want to provide a value for.

```
>>> testPars( two|"some value" )
one: Nil; two: some value
```

The pipe "`|`" allows to name directly a parameter for which a value is given; other unnamed parameters are filled with nil. Both conventions can be merged in the same call:

```
>>> testPars( two|"some value", "first" )
one: first; two: some value
```

Initially, all of the unnamed values are used to fill the parameter list, as if the named values did not exist. Then the named values will be assigned to the list, possibly replacing parameters assigned by unnamed values.

One interesting aspect of the named call convention is that the "`|`" pipe operator isn't simply a grammar sugar seeking the parameter in the function definition. It creates what is called *future binding*. Similarly to late bindings, which take a value only when evaluated in a functional context or in a array call, future bindings receive a value that is kept dormant until they are found relevant (i.e. in function calls). Also, while the left part of the pipe expression must be a symbol, the right part can be any Falcon expression.

For example:

```
>>> param = two | "a" + " " + "composed" + " " + "string"
: &two
>>> testPars( param )
one: Nil; two: a composed string
```

The future binding stored in the variable has been diligently applied in the function call. And of course, this also works in delayed array calls:

```
>>> v = .[testPars param]
: Array
>>> inspect(v)
Array[2]{
   Function testPars
   &two
}
>>> v()
one: Nil; two: a composed string
```

To obtain the benefit of the implicit variable parameters call convention, we need some core functions that will tell us about the parameters that have been currently passed to our function. The function `paramCount` returns the number of parameters with which the caller function has been called, and the function `parameter` returns the nth positional parameter, the first one begin numbered 0.

The following example takes the first parameter as fixed; other parameters are then printed in a loop.

```
>>> function xprint( prompt )
...    for p in [1 : paramCount()]
...       > prompt, parameter( p )
...    end
... end
>>> xprint( "**", "one", "two", "three" )
**one
**two
**three
```

The for/in loop in this example counts from and including 1 up to but excluding the number returned by `paramCount`. It's a "ranged" for/in loop, and the `[x:y]` notation creates a range.

Example:

```
>>> range = [0:2]
: [0:2]
>>> for number in range: > number
0
1
```

Classes and objects

At last, some OOP. Well, if it's OOP that you want, Falcon will give you even more than you'd expect. In fact, classes are just one of the means by which Falcon provides OOP constructs; we have more. Other than classes, Falcon knows the concepts of singleton objects, blessed dictionaries, tables and array (OOP) bindings. We're seeing the classes and objects here; we'll leave the somewhat more creative concepts of tables and bindings for the next chapter.

Classes are types of similar items, having some common data structure (properties) and operations that can be performed on that data (methods). A live closed set of data and methods is an instance. In Falcon, classes can also have attributes, that is, common inter-class binary properties which denotes a special "belonging" of classes, and their instances, to common logical sets.

A class is declared in the following way:

```
>>> class Vector
...    x = 0
...    y = 0
...    z = 0
...
...    function  module()
...       return ( self.x * self.y * self.z ) ** (1/3)
...    end
... end
```

Assigning a variable inside a class block creates a property; declaring a function inside a class creates a method. Actually, as Falcon doesn't differentiate between data and functions, callable items (as functional sequences or nameless functions) can be assigned to properties, both immediately or after the item is created, actually making them methods, and methods can receive plain data later on, becoming just properties. So, the role between properties and methods can be dynamically exchanged.

As shown in the `module` method, the `self` keyword can be used to access properties (and so, methods) in the current instance of a class. The `**` (double times) operator performs a power; in our case, it performs a cube root.

An instance is created simply by calling the class name:

```
>>> v = Vector()
: Object
>>> inspect(v)
Object of class Vector {
   module => Function Vector.module
   x => int(0)
   y => int(0)
   z => int(0)
}
```

We may then access properties in the object using the "dot accessor":

```
>>> v.x, v.y, v.z = 10, 5, 4
: Array
 >>> v.module()
: 5.848035476425731
```

Notice that we have used the short distributive assignment notation to assign the three values of `x`, `y` and `z` properties in one statement.

Class declaration can include some initialization parameters and an initialization block (a sort of constructor). Properties can be assigned any expression, which are evaluated at instance creation, right before processing the `init` block. To demonstrate:

```
>>> class XVector( x, y, z )
...     x = x
...     y = y
...     z = z
...     magnitudo = self.module()
```

```
...
...    init
...       > "Just created a vector of magnitude ", self.magnitudo
...    end
...
...    function module()
...      return (self.x * self.y * self.z) ** (1/3)
...    end
... end
>>> xv = XVector( 5, 6, 7 )
Just created a vector of magnitude 5.943921952763129
: Object
```

Of course, properties need not to be named after parameters, but, as in this example, it is possible and often sensible to do so. Evaluation of expressions initializing the properties is granted to be performed in first-to-last order, and properties in the same class can be accessed with the self keyword (so they can be distinguished from parameters); also, all the methods declared with the `function` keyword are granted to be created and readied before the properties are initialized; so, it is possible to call methods in property initialization as in the above example.

Method persistence

One very important characteristic of Falcon OOP is what is called “method persistence”. As we said, we can put functions in every instance member, even in what was previously a property, making that a method. See the following example:

```
>>> class Varying
...   data = "some data"
...   var = nil
... end
>>> x = Varying()
: Object
```

Now that we have an instance, we can ascertain its property as being a real property...

```
>>> x.var
: Nil
: Function _lambda#_id_1
```

... then turn it into a method ...

```
>>> x.var = function(); > "My data is ", self.data; end
```

... and finally call it.

```
>>> x.var()
My data is some data
```

Also, it is possible to assign functional sequences and statically declared functions. In every case, the `self` item will take the value of the instance in for which the call is performed.

This is so important that it is granted even for calls at a later time. It is possible to store methods safely elsewhere and then call them:

```
>>> method = x.var
: Method _lambda#_id_1
>>> x.data = "other value"
: "other value"
>>> method()
My data is other value
```

Once created, there isn't any way to separate a method from its original instance. It is even possible to store the method in another instance, but its `self` item will refer to the first one. As an example:

```
>>> y = Varying()
: Object
```

```
>>> y.var = method
: Method _lambda#_id_1
>>> y.var()
My data is other value
```

As you can see, “other value” was taken from x, the original instance where the method comes from, and not from y, the instance in which it has been placed at a second time. In fact,

```
>>> inspect(y)
Object of class Varying {
   data => "some data"
   var => Method 0x80914E0->_lambda#_id_1      Object of class Varying {
         data => "other value"
         var => Function _lambda#_id_1
      }
      Function _lambda#_id_1
   }
}
```

The `inspect` function reports var as not being a plain function, but as a method referencing the other instance.

And of course, even taking that method back from instance `y`, it will still refer to `x`:

```
>>> mth2 = y.var
: Method _lambda#_id_1
>>> x.var = "still changed"
: still changed
>>> mth2()
My data is still changed
```

This aspect of Falcon OOP allows functional sequences to be built which call methods with cached parameters. For example, we can build quite an interesting functional loop involving an instance:

```
>>> class Funarr
...    data = .[ 'one' 'two' 'three' 'four' ]
...    function show( id )
...       > "Showing item ", id, ": ", self.data[id]
...    end
... end
>>> x = Funarr()
: Object
```

We have our instance. Notice that the strings in the array are declared using single quotes (`'`). Single quote strings are not chained, while double quote strings get merged if they are declared sequentially and separated only by whitespace. As we want four separate strings as array elements, this notation allows for a bit more compact declaration.

Then we can use our method.

```
>>> showmeth = .[x.show &i]
: Array
>>> .[times x.data.len() &i .[ showmeth ]]()
Showing item 0: one
Showing item 1: two
Showing item 2: three
Showing item 3: four
: 4
```

Remember the name of variables prefixed with '`&`'? They are called *late bindings*. The `&i` late binding sleeps in the `showmeth` array until used in a functional evaluation or array call; so it gets the value given to the same `&i` as it gets given a value by the `times` function. You can also see the `x.show` method keeping track of the source object, and using that to retrieve the needed data.

Inheritance

In Falcon, you can derive new classes from previously declared ones. This means that you can extend the functionalities of simpler classes by adding new pieces to them. See this example:

```
>>> class Base
...     pbase = 'base property'
... end
>>> class Derived from Base
...     pnew = 'derived property'
... end
>>> x = Derived()
: Object
>>> x.pnew
: derived property
>>> x.pbase
: base property
```

The from keyword declares a set of inheritances. It is not necessary for the base classes declared in the from clause to be already known; they could be declared later on in the same file, or come from foreign modules.

Falcon also supports the concept of multiple inheritance; new classes can take base methods and properties from more base classes. Declared subclasses are instantiated in the same order they are declared in the from clause, and properties with the same name coming from the upper classes overwrite lower properties.

For the following example, exit and reenter Falcon's interactive mode, as we will be using a previously defined symbol:

```
>>> class Base1( val )
...     p1 = val
...     init
...        > "Hello from base1"
...     end
...     function mth()
...        > "Method from base1"
...     end
... end
>>> class Base2( val )
...     p2 = val
...     init
...        > "Hello from base2"
...     end
...     function mth()
...        > "Method from base2"
...     end
... end
>>> class Derived( v1, v2 ) from Base1( v1 ), Base2( v2 )
... end(
>>> x = Derived( "one", "two" )
Hello from base1
Hello from base2
: Object
>>> x.p1
: one
>>> x.p2
: two
>>> x.mth()
Method from base2
```

As you can see, we have initialized the two base classes by passing them the parameters received by the upper class; it is possible to use any expression made of parameters and constants as base-class initializers. Initialization is performed in first-to-last order, and classes declared later in the from clause can overwrite and undo what done in previously declared classes. The derived objects has the p1 and p2 properties, correctly initialized, coming from both base classes, but the method `mth` is coming from the class that declared it last. However, it is still possible to access the method from Base1:

```
>>> x.Base1.mth()
Method from base1
```

Similarly, the derived class may define new methods or overload methods and properties from previous classes.

Singleton objects

Usually, Falcon is not very interested in knowing the exact class to which instances belong. As long as they provide the needed properties, you can use instances of different classes interchangeably.

And you can even create objects without classes.

The object declaration works exactly as the class declaration, but it defines a single instance which is immediately readied (actually, it is readied before the VM executes the main code of the program).

Objects can even derive from base classes; they will inherit and overload base classes methods and properties as it's done for normal class inheritance, but they themselves won't be considered as instances of a particular class.

For example, we may create an object like the following:

```
>>> object Alone
...     name = "alone"
...     function hail()
...       > "I am ", self.name
...     end
... end
>>> Alone.hail()
I am alone
>>> Alone.name = "happy"
: happy
>>> Alone.hail()
I am happy
```

Objects cannot have initialization parameters, but they can have an `init` block.

Blessed dictionaries

As with many modern scripting languages, Falcon has dictionaries. We haven't seen them yet, and won't see them in detail here, as they are part of that boring stuff that we want to avoid. But just to give you an idea, they work like this:

```
>>> dict = [ "key" => "value", "another key" => "another value" ]
: Dictionary
>>> printl( dict[ 'another key' ] )
another value
>>> for k, v in dict
...     > k, " => ", v
... end
another key => another value
key => value
```

Keys and values can be anything, including other dictionaries. However, the most common key types are probably numbers and strings.

Applying the function `bless` on a dictionary, it becomes a kind of light object, still maintaining its dictionary status, which can be accessed through the “dot” operator, and that can have functions which become methods as they are extracted.

```
>>> dict = bless( [ "prop"=>100, "calc"=>(lambda x=> x * self.prop)] )
: Dictionary
>>> dict.calc(3)
: 300
>>> dict["prop"] = 200
: 200
>>> dict.calc(3)
: 600
```

Notice how we have mixed the dictionary accessor [] with the dot OOP accessor in the above example.

As for classes and objects, also dictionary methods can be cached:

```
>>> mth = dict.calc
: TabMethod _lambda#_id_2
>>> dict.prop=300
: 300
>>> mth(3)
: 900
```

Array bindings

We have already seen them, but it may be worth noticing how array bindings can be mixed with late bindings to have flexible functional constructs that can be configured on the fly.

For example, see the following code:

```
>>> hail = .[printl 'Welcome, ' &prefix ' ' &name '!']
: Array
>>> hail.prefix = "Mr."
: Mr.
>>> hail.name = "Ford"
: Ford
>>> hail()
Welcome, Mr. Ford!
```

But also, by assigning a function to a binding:

```
>>> hail.tell = function(); printl( @ "Hey, $self.prefix $self.name, did you know..."); end
: Function _lambda#_id_1
>>> hail.tell()
Hey, Mr. Ford, did you know...
```

Notice that we have accessed the self object through the string expansion operator.

Naturally, the self object in binding methods reflects the owning array, and can then be seen as an array...

```
>>> hail.inspect = function(); for item in self; >item; end; end
: Function _lambda#_id_2
>>> hail.inspect()
Function printl
Welcome,
&prefix

&name
!
```

... or even as a functional sequence as it was originally meant:

```
>>> hail.do = function(); self.name = "Smith"; self(); end
: Function _lambda#_id_4
>>> hail.do()
Welcome, Mr. Smith!
```

All this flexibility can be a bit disorientating, and it takes sometime to get accustomed to it, but once you grasp it, you won't let it go. Yet, we're not done; we're still a couple of steps behind what's coming next.

Tabular programming

Long before fourth generation languages appeared, long before compilers appeared, even long before computers ever appeared, there was a time in which people needed to categorize things efficiently and effectively into simple, easily understood, and useful groups.

Tables have been the most effective categorization tool for hundreds years, and they continue to be useful for marketing and strategy gurus. Compared to tables, the formal definition of "class" as the base of object oriented programming is relatively young. Outside of IT labs, people use tables to analyze reality and drive their decisions. Not surprisingly, every managerial decision making system involves some sort of grid, or table, that can just as easily be built with advanced instruments as with pencil & paper, and can drive decisions worth billions of dollars.

Despite this, IT has relegated tables to the ancillary role of storing data. *Lots* of *interesting* data, for sure, but still just data.

Falcon can employ tables to drive program logic as they can drive decision making systems.

Table are classes

The base of tabular programming is the `Table` class, which is simply a normal class except for some very special interactions with arrays (handled transparently). It would be more correct to say that arrays know what a `Table` instance is, and how to interact with them, than saying that the `Table` class is *special*.

More precisely, a `Table` is a class storing a set of rows, each of which is an array, and a heading which describes the meaning of each column. Tables can have one or more pages, that is, sets of rows that can be accessed at a time, and each heading can optionally be provided with "column data", whose semantic value can vary depending on the operations being performed. Both row and column data can be any Falcon object, including other tables, while column names are limited to strings which ideally do not contain whitespace.

```
>>> x = Table( [ "name", "income" ],
... [ 'Smith', 2000 ],
... [ 'Jones', 2800 ],
... [ 'Sears', 1900 ],
... [ 'Sams',  3200 ] )
: Object
```

We have just created a table, which is a standard object, with a mandatory first parameter (the column heading) and a set of rows (each one in a different parameter).

Rows can be then accessed through the get method, which returns an array (the row).

```
>>> inspect( x.get(0) )
Array[2]{
   "Smith"
   int(2000)
}
```

Falcon sees tables as sequences, so it is possible to iterate on each row:

```
>>> for l in x
...     > @ "name: $(l[0]:r10)   income: $(l[1]:r6)"
... end
name:      Smith   income:   2000
name:      Jones   income:   2800
name:      Sears   income:   1900
name:       Sams   income:   3200
```

The "`:r10`" and "`:r6`" are format specifiers, and they simply right justify and pad the value.

Arrays stored in tables still are accessible from outside of the table. To demonstrate:

```
>>> row = [ 'Benson', 2300 ]
```

```
: Array
>>> x.insert( 0, row )
>>> row[1] = 2450
: 2450
>>> inspect( x.get(0) )
Array[2]{
   "Benson"
   int(2450)
}
```

They also assume two important properties. First, they become immutable; their size stays fixed to the number of columns in their table (which can vary later on); yet, it is possible to change each element, as long as this doesn't shrink or grow the array. Second, they inherit the table columns, which references their entries. In other words, the `row` variable of the example above knows that it is part of a table, and that it's first element can also be called “name”, while the second can be called “income”.

```
>>> row.name
: Benson
>>> row.income = 2500
: 2500
>>> inspect( row )
Array[2]{
   "Benson"
   int(2500)
}
```

Like any other array, table rows can have also bindings; the main difference between the bindings and the table column names is that bindings lay *besides* the array, while table names lie *inside* it, and allow indirect access of their ordinal content. This allows the array to be virtualized as an accessible object when needed, while accessing it with a direct index for when higher performance is necessary.

As for pure bindings, methods extracted from the array by their column names can refer to the `self` item;

```
>>> row.income = function(); return self.name.len() *800; end
: Function _lambda#_id_1
>>> row.income()
: 4800
```

and they can also refer to the table they come from:

```
>>> row.income = function(); return self.table().get( self.tabRow()+1 ).income + 100;end
: Function _lambda#_id_2
>>> row.income()
: 2100
```

The code above refers the table and the row in the table at which our `self` is placed. The `table` and `tabRow` methods are pre-defined methods of the *Falcon Basic Object Model*. The FBOM are a set of methods, some common to all the Falcon items, while others are specific to certain item types, which can be generally applied to any Falcon item, including numbers, strings and even `nil`.

Default Values

As said previously, tables can be provided with *column values*. They can be added at initialization using future bindings in the heading entry:

```
>>> table = Table(
...    [ name|"unknown", income| lambda => self.name.len()*100 ],
...    [nil, nil],
...    [ "Smith", nil ],
...    [ "Sams", 2500 ] )
: Object
```

This created a table with two columns, each having an associated column value.

```
>>> for row in table
```

```
...       > row.name, ": ", valof( row.income )
... end
unknown: 700
Smith: 500
Sams: 2500
```

They can also be accessed directly through the `columnData` method;

```
>>> table.columnData( 0 )               // access
: unknown
>>> table.columnData( 0, "known" )      // change
: unknown
>>> table.get(0).name
: known
```

Table operations

Other than providing names and defaults to access row elements, tables have a set of logical operations that can be used to perform code selection and branching.

The choice method feeds all the rows in the table to an external function for evaluation; the row that gets the highest value is then selected and returned. For instance, suppose that we have to pick the discount function to be applied to a certain customer. The next sample program is a bit complex, so it's better to save it in an editor and execute it from the command line:

```
offers = Table(
   .[ 'base' 'upto'    'discount' ],
   .[     0   2999     lambda x=>x],
   .[  3000   4999     lambda x=>x*0.95],
   .[  5000   6999     lambda x=>x*0.93],
   .[  7000      0     lambda x=>x*0.9])

function pickDiscount( pz, row )
   if pz >= row.base and ( row.upto == 0 or pz <= row.upto )
      return 1
   else
      return 0
   end
end
```

The `pickDiscount` function will receive a `pz` parameter needed for configuration and a row; the row is the element in the table that the choice method extracts and feeds into `pickDiscount`, Then, if the value of the `pz` variable is between the base and upto items in the row, 1 is returned; in all the other cases, 0 is returned. This means that the function will select that row where our value lies.

The following statement performs an evaluation, passing 3500 as the `pz` value, and then applying the discount lambda of the row that applies to an arbitrary price.

```
> offers.choice( [pickDiscount, 3500 ] ).discount(500)
```

We can build an interesting "all in one" function as follows:

```
offer = .[offers.choice .[pickDiscount &qty ] 'discount' ]
```

We created a functional sequence made of the choice method on the offer table and the callable it should receive to pick the desired row. The callable itself uses a `qty` binding that can be configured later on; then, instead of accessing the discount column via the dot accessor (or using the `at` function), we can use the extra parameter of the `choice` method, which designates a single column value to be returned. Calling this `offer` function, we automatically select the discount that is to be applied to customers having a ranking as designed by the qty binding.

This may be used like the following:

```
offer.qty = 500
> "Price 500 discounted for a small customer: ", offer()(500)

offer.qty = 3200
> "Price 500 discounted for a medium customer: ", offer()(500)

offer.qty = 6300
> "Price 500 discounted for a big customer: ", offer()(500)
```

The program runs with this result:

```
Price 500 discounted for a small customer: 500
Price 500 discounted for a medium customer: 475
Price 500 discounted for a big customer: 465
```

The advantage of using a table for data definitions instead of switches, a cascade of nested ifs, method overloading in class hierarchies, and so on is that the parameter definitions may be changed live, as the program runs. A new row may be inserted, a new column may be added, the discount conditions or the level limits may be altered, and still our offer functional sequence would reflect the up-to-date status of the condition table.

Also, an interesting feature of tables is that of being able to swap all its data at once, through pagination.

Table pages

Tables can have an arbitrary number of pages which can be independently grown, shrunk, changed and generally updated. All the pages share the same column structure and column data; only the rows are changed. So, inserting, removing and changing columns is immediately reflected on every row of every page. Each page can have a different size, and rows extracted from a page are also available when switching the active page.

Activating a page will have the effect of causing `get`, `find`, `insert`, `remove` and other table-wide operations to be performed exclusively on the current page.

For example, suppose there is a set of bank account meta-data which has the interest rate for a certain account category. In this case, a table may store all the account types and their interest rates in different pages, that can be marked at different times.

The following class extends the base `Table` to select a page depending on the year for which calculations are required.

```
object AccTable from Table( [ "name", "rate_act", "rate_pasv" ] )

   init
      // in year 2007
      self.insertPage( nil, .[
         .[ 'basic' 2.3 8.4 ]
         .[ 'business' 0.8 4.2 ]
         .[ 'premium' 3.2 5.3 ]
         ])

      // in year 2008
      self.insertPage( nil, .[
         .[ 'basic' 3.2 7.6 ]
         .[ 'business' 1.1 4.5 ]
         .[ 'premium' 3.4 5.2 ]
         ])
   end

   // set the current year
   function setYear( year )
      if year < 2008
         self.setPage(1)
      else
         self.setPage(2)
      end
   end
   // get rates
   function activeRate( acct, amount )
      return amount + (self.find( 'name', acct ).rate_act / 100.0 * amount)
   end
end

// A bit of calculation

// what should a premium account get for $1000 on 2007?
AccTable.setYear(2007)
> "Premium account with $1000 in 2007 was worth: ", \
   AccTable.activeRate( 'premium', 1000 )

AccTable.setYear(2008)
> "Premium account with $1000 in 2008 was worth: ", \
   AccTable.activeRate( 'premium', 1000 )
```

Via the `setYear` method, we have been able to change the underlying data definition and transparently use new calculation parameters. This also could have been easily done using simpler methods, and could definitely be done using OOP; but suppose that it's not just the parameter to change in years, but the whole logic that regulates refunds of credit accounts.

Suppose that starting in 2008, the bank starts to add a fixed commission of $50 for accessing credit lines.

```
object AccTable from Table( [ "name", "rate_act_func", "rate_pasv_func" ] )

   init
      // in year 2007
      self.insertPage( nil, .[
         .[ 'basic'      lambda x => x * 2.3/100     lambda x => x * 8.4/100 ]
         .[ 'business'   lambda x => x * 0.8/100     lambda x => x * 4.2/100 ]
         .[ 'premium'    lambda x => x * 3.2/100     lambda x => x * 5.3/100 ]
         ])

      // in year 2008 — notice: we use comma because x ( y ) would mean a call
      self.insertPage( nil, .[
         .[ 'basic', (lambda x => x * 3.2/100), (lambda x => x * 7.6/100+50) ]
         .[ 'business', (lambda x => x * 1.1/100), (lambda x => x * 4.5/100+50) ]
         .[ 'premium', (lambda x => x * 3.4/100), (lambda x => x * 5.2/100+50) ]
         ])
   end

   // set the current year
   function setYear( year )
      if year < 2008
         self.setPage(1)
      else
         self.setPage(2)
      end
   end

   // get rates
   function creditCost( acct, amount )
      return self.find( 'name', acct ).rate_pasv_func(amount)
   end
end

// A bit of calculation

// what should a premium account get for $1000 on 2007?
AccTable.setYear(2007)
> "Asking for $1000 in 2007 costed: ", \
   AccTable.creditCost( 'premium', 1000 )

AccTable.setYear(2008)
> "Asking for $1000 in 2008 costed: ", \
   AccTable.creditCost( 'premium', 1000 )
```

The output of this program is:

```
Asking for $1000 in 2007 costed: 53
Asking for $1000 in 2008 costed: 102
```

Doing the same thing with traditional OOP constructs would have required the program to account for this variability from the very beginning of the project, or face the eventuality that the classes initially designed to represent bank accounts will be missing just flexibility needed to implement the last twist that the marketing guys have thought to make their bank sexier.

OOP is very flexible, and Falcon adds tons of flexibility to the base model already, providing rich functional constructs, prototype oriented OOP and so on. But at times, this isn't just enough, and thinking of some problem categories as two-dimensional tables (or eventually as three-dimensional tables-in-time as in this example) better fits the de-structured and pragmatic approach to problem definition and analysis that business professionals are accustomed to using (which they send down to the development department for implementation).

Table-wide operations

For brevity, we'll consider only one significant operation taking the whole contents of the (current page of a) table.

Other operations are described in the Table class reference, and exemplified in other manuals.

The `bid` method selects a row given a column. The rows must provide an evaluable item (i.e. a function) at the specified column; the item is called in turn, and must return a value greater than zero. At the end of the bidding, the item offering the highest value will be selected.

Consider the following simulation: several algorithms are struggling to win the highest possible number of auctions out of N (a number known in advance). One algorithm bids a fixed amount, another bids a random amount and a third bids a base price doubling it each time if it doesn't win. After each bidding round, the bid amount is removed from a pool of resources initially given to each bidder.

The first two algorithms do not have any state, so they are quite easy to code:

```
// a function always betting the same value
function betFixed()
   return self.table().amount / self.table().turns
end

// a function always betting a random value
function betRandom()
   return self.table().amount * random()
end
```

The last algorithm must remember its previous bet, and if it was a winning bet or not. To handle this, we'll have a state property, called `storage`, where the bid is placed, and a property filled by the calling table indicating if the algorithm won in the previous turn or not. As this information may be interesting for every bidder, it will be placed in a table column called `hasWon`. Each turn, this column will be reset and the winner row will have this value set.

```
// a function betting each time the double of the previous time
function betDouble()
   if self.hasWon
      bid = self.table().amount / self.table().turns / 2
   else
      if self provides storage
         bid = self.storage * 2
      else
         bid = self.table().amount / self.table().turns / 2
      end
   end

   self.storage = bid
   return bid
end
```

We'll see first how the algorithm works without the help of tabular programming, and then see how table operations can be employed to simplify it.

The table heading is like the following:

```
class BidTable(amount, turns) \
      from Table( [ "name", "bet", "amount", "hasWon", "timesWon" ] )
   amount = amount
   turns = turns
```

A simple utility method allows us to create our rows:

```
   function addAlgorithm( name, func )
      self.insert( 0, [name, func, self.amount, false, 0] )
   end
```

The betting process involves calling all the algorithms, recording what they bet, provided they can pay using what's left of their initial account, and removing the bet quantity from the accounts. Then, the algorithm having bet the highest value is declared to be the winner:

```
   function bet( time )
      > "Opening bet ", time+1, ": "

      winning = nil
      winning_bid = -1
```

Each row is taken in turn for betting:

```
      for row in self
         bid = row.bet()
         if bid > row.amount: bid = row.amount
         > @"  $(row.name) is betting $(bid:.2) out of $(row.amount:.2)"
         row.amount -= bid
```

Then, if this bet is better than the others, it is recorded as the temporary winner:

```
         if bid > winning_bid
            winning = row
            winning_bid = bid
         end
      end
```

Finally, it is necessary to declare the winner; to do this, we must scan the entire table and set the winning flags correctly for each participant:

```
      // declare the winner
      for row in self
         if row == winning
            winning.hasWon = true
            winning.timesWon ++
            > "   ", row.name, " wins this turn!"
         else
            winning.hasWon = false
         end
      end
   end
```

The game consists of playing the bet stage for the required amount of times, and picking up the final ranking.

```
   function game()
      for i in [0:self.turns]
         self.bet(i)
      end
      > "===================================="
      > "           Final Ranking            "
      > "===================================="
```

We can use the `arraySort` function to sort the algorithms taking into account the times they have won. The `getPage` table method will take a copy of the page as it is now as an array containing all the rows, and that can be sorted leaving the actual data in the page untouched. The sorting just requires a function returning 1, 0 or -1 depending on the order that needs to be applied; the `compare` FBOM method applied on the `timesWon` table property will do the trick.

```
      bidders = self.getPage()
      arraySort( bidders, lambda x, y => -x.timesWon.compare( y.timesWon ) )
      for id in [0:bidders.len() ]
         >> @ "[$(id)] $(bidders[id].name) with $(bidders[id].timesWon) victor"
         > bidders[id].timesWon == 1 ? "y" : "ies"
      end
   end
end
```

There is nothing else to do but fill the table and start the game:

```
randomSeed( seconds() )
```

```
bt = BidTable( 100, 6 )

bt.addAlgorithm( "Mr. fix", betFixed )
bt.addAlgorithm( "Random-san", betRandom )
bt.addAlgorithm( "Herr Double", betDouble )

bt.game()
```

The method `bidding` of the `Table` class does more or less what the bidding function we have created does in a table: it calls a method stored in a column, recording the one that returned the highest value and returning it. However, the loop in the bid method of this sample is not just selecting the row with the algorithm returning the highest value; it also changes the status of each row. Because of this, we'll need an extra column to store a method that does some house cleaning before and after the call of the betting algorithms:

```
function genericBetting()
   bid = self.bet()
   if bid > self.amount: bid = self.amount
   > @"  $(self.name) is betting $(bid:.2) out of $(self.amount:.2)"
   self.amount -= bid
   return bid
end
```

We may store this generic cleanup routine in another column, as column wide data, so that it will be used if the corresponding cell is nil when the bid is performed:

```
class BidTable(amount, turns) \
      from Table( [ "name", "bet", "amount", "hasWon", "timesWon", genbid|genericBetting ] )
   amount = amount
   turns = turns

   function addAlgorithm( name, func )
      self.insert( 0, [name, func, self.amount, false, 0, nil] )
   end
```

Notice the `genbid` column, which is given nil in `addAlgorithm`.

Now the bet function can be simplified:

```
   function bet( time )
      > "Opening bet ", time+1, ": "
      winner = self.bidding( 'genbid' )
      > "  ", winner.name, " wins this turn!"
      winner.timesWon ++
```

Instead of clearing all the `hasWon` properties during the `genbid` calls, and then setting the value for the winner, we use this method which does the same thing in one step:

```
      self.resetColumn( 'hasWon', false, winner.tabRow(), true )
   end
```

The rest of the program is unchanged.

The method `choice` is similar to bidding, but it calls a generic function provided during the call. As `choice` calls the given function passing it one row at a time, we have to change each reference the generic betting function into:

```
function genericBetting( row )
   bid = row.bet()
   if bid > row.amount: bid = row.amount
   > @"  $(row.name) is betting $(bid:.2) out of $(row.amount:.2)"
   row.amount -= bid
   return bid
end
```

Then, just change the `self.bidding` call to

```
winner = self.choice( genericBetting )
```

Also, it is now possible to remove the extra column “`genbid`”.