THE FALCON'S SURVIVAL GUIDE

Table of contents

Introduction

Falcon is what is commonly known as a scripting language ; program source code is simply referred as the "script". The script is organized into lines that are executed one after another (empty lines being ignored). Every line is interpreted as a command, or more properly, as a statement. The most basic statement is the assignment , which allows storing a value into a variable. A variable is exactly that: a temporary storage for a value. Meet our first falcon script:

```
number = 0
```

This is an assignment storing the value `0` into the variable named `number` . Variable names must start with a lower case or uppercase letter, with a _ underline sign or with any international character; after that you may use any letter or number, and the "_" underline sign. Some examples of valid names: `number`, `_Aname`, `variable_number_0`.

Falcon understands symbol names in any language of the world. For example , the number we have seen above may be written in Chinese:

```
数 = 0
```

Falcon provides three elementary value types: integer numbers, floating point numbers and strings. Integer numbers include 0, 1, 2 and also -1, -2, -3 and so on. Floating point numbers are numbers that contain a floating point; i.e. 1.2, 0.001, or scientific notation numbers such as 1.34e15, indicating 134 followed by 13 zeros. Falcon also supports hexadecimal and octal numbers in standard C notation (0x$_{\text{. and }}$0·); explaining what hexadecimal and octal numbers are is beyond the scope of this text (and if you don't know what they are, then you don't need them).

Strings are sequences of characters enclosed with double quotes, like this: `"this is a string"` . Strings may span across multiple lines; and in that case any blanks or tabs before the first character will be removed.

An element that is quite important in every programming language is the comment. Comments are pieces of text that are completely useless to the program, but that can be used to write information, warnings or just notes in the text. Falcon supports C-language style single line and block comments. A single line comment begins with two slashes (//) and ends at the end of the line. A block comment begins with a slash followed by an asterisk, and terminates when an asterisk followed by a slash is met (/*...*/). Many programmers love to mark code areas with lots of asterisks or slashes. So, we are ready to see how a Falcon script may look like:

```
/*********************************
   My First falcon Script
*/

var1 = 156                  // An integer number
var2 = 1e15                // A scientific notation number
var3 = "Hello World"    // A string

///////////////////////////// end of the script
```

Falcon supports the "execute script" pre-processing directive for Unix shells: if the first line of the script begins with a pound sign (#) the line is ignored by Falcon . Its

> *use is to indicate to the shell that it's the Falcon interpreter that is to execute the script, i.e. `#!/usr/bin/falcon`*

Other than comments, statements and elementary constants, the last Falcon basic brick is the expression . An expression is a set of elementary constants, variables or simpler expressions connected via operators. Here are some examples:

```
number = 10
sum = number + number
value = number * sum * 5 + 2
complex_value = value * (number + 1.2 ) - 15 * (sum/3.15)
sum_of_string = "A string" + " "+ "and another"
```

Falcon provides a number of operators that work on strings. For example , summing a string to another, or to a variable which contains a string, will result in new strings containing the two original strings joined together.

The mathematical operators that Falcon supports are addition (+), subtraction (-), multiplication (*), division (/), power (**) and modulo (%). The power operator can take a fractional number as a second operand; in this way it's possible to perform arbitrary roots. For example , 100 ** 0.5 will give 10 as result. The modulo operator can be applied to two integer numbers ONLY and gives the remainder of their division; for example , 10 % 5 is 0, while 10 % 3 is 1 (because 3*3 = 9, and 10 - 9 = 1). Applying the modulo operator to non-integer numbers will cause an error to be raised.

Falcon also supports "self operators"; they are +=, -=, *=, /=, **= and %=. Self operators assign to a variable the operations they represent using the same variable value in the operation. For example:

```
number = 10
number += 1
```

will cause `{number}` to become 11.

> *Falcon also provides ++ and -- operators. Both prefix and postfix operators are supported.*

One particularly important expression in Falcon is the function call . Function calls are composed of a symbol near two round brackets which can contain a list of parameters , each of which may be an expression and separated by commas. Like this:

```
number = 10
printl( "The square of 10 is: ", number * number )
```

We just met our first function: `printl`. This function prints all of its parameters and then advances the cursor to the next line. Another function, named `print`, just prints all the parameters without advancing the cursor to the next line.

> *`print` and `printl` are actually using the virtual machine standard output stream to perform output. This stream can be managed by the embedding application, redirected to files, and encoded through Unicode or through other encodings.*

Functions may "return a value"; `print` and `printl` do not return any value, but a function that returns a value may be used as a part of an expression, like this:

```
number = 10 * a_function_of_some_kind( 10 )
```

As variables, function names may be written in any language. For example, the following sentence in Japanese is correctly understood in Falcon:

```
かず = 10 * 例え_ファンクチオン( 10 )
```

Your first Falcon script

It's time to execute your first Falcon script. To do this, write the following lines in a text editor (i.e. Notepad), save the file as first.fal and enter a console (also known as "Ms-Dos" prompt, or "cmd" prompt). To do this, you are supposed to know how to access the console and change the directory. Minimal survival instructions for Windows-users are: press "start" and select "execute command". If you are running Windows 95, 98, or ME write "command" in the box that appears on the screen; if you are running Windows NT, 2000 or XP write "cmd". Press enter, and voilà. Under Linux, for example, just open up a Terminal. Finally, launch Falcon with the command "falcon first.fal". Here is the code for first.fal:

```
/************************************
* First Falcon program: first.fal
* This is just a comment; its content
* does not matter, is just here to
* remind you that this was your first
* Falcon script ever.
************************************/

> "Hello world."
```

When a line begins with > what follows is sent to the output stream and a newline character is added. This is called called fast print, as opposed to other methods to write output (for example, through the `printl()` function that has been shown before).

A line may begin with >> alone, in which case what follows is sent to the Falcon output stream, but without appending a final end-of-line (EOF) character.

Control structures

Falcon programs, or scripts, are executed one line after another, beginning from the first to the last. This is a quite dull way to program, so it's possible to modify the order by which lines are executed. This is done using some statements that go generally under the name of "control structures", and are divided in two main categories: conditional control structures and loop control structures. We'll begin with the former.

Conditional structures allow the execution of part of the script if a condition is verified, or otherwise acknowledged to be true. Verifying that a condition is true is the responsibility of the "relational" operators, and by their big brothers the "logical operators". Relational operators put two variables or values in a relation and the expression becomes either true or false depending on the truthfulness of the expression. The most important relational operators are "==", ">", "<", ">=" (greater or equal), "<=" (less or equal) and "!=" (different from). Note that "==" is used in place of "=", as the latter is used for assignment.

The if/elif/else statement

The basic control structure is the `if` statement. The `if` statement executes a set of statements

beginning with the very next line up to the "end" statement only if the condition is true. Look at this example:

```
number = 10

if number > 10
   printl( "This is impossible" )
end
```

As the expression above is always false, the `printl` function will be never called. The "if" statement may have an optional "else" clause that is executed only when the expression is false. Look at this example:

```
number = 10
if number > 10
   > "This is impossible"
else
   > "This is possible"
end
```

This code will always print "this is possible". Finally, the `if` statement may contain a list of clauses called `elif`. Every `elif` clause evaluates an expression; if the expression is true, then the instructions below the `elif` are executed, else the next `elif` is checked, or else block is executed. This is how a complete if statement should look:

```
if expression
    statements...

elif expression
    statements...

elif expression
    statements...

   /* other elifs */
else
    statements...

end
```

Empty lines are always ignored; putting space below every statement block is considered "elegant". Also, it's wise to put some space right before every statement that is inside a block; all this makes immediately visible what a block is supposed to do and where it is supposed to end. The technique of moving the statements inside a block is called indentation and is considered very important for program readability; having a good indentation style will make your code be regarded as clean and professional. As a general rule, add three spaces each time you put some statement inside a code block.

Let's see a more interesting example for the `if` statement:

```
print( "Enter your age: >" )
age = int( input() )

if age <= 5
   > "You are too young to program. Maybe."

elif age > 12
   > "You may already be a great Falcon programmer."

else
   > "You are ready to become a programmer."

end

> "Thanks for telling me"
```

Try this program and enter some numbers. Then try to enter a letter; you'll get what is called a "runtime error" that will terminate your application. We'll learn how to take advantage of this fact later; for now go on trying some numbers.

The `input` function will read the data you type up to when you press the enter key; the `int()` function will try to convert your typing into an integer number. Now look at the `if`; the statements inside it are executed only when the entered number is equal or less than 5. If this happens, neither the following `elif` nor the `else` blocks are executed; if the number is greater than five then the `elif` condition is checked. If it's true, the statements inside it are executed and the else block is skipped; if it's false, the else block will finally be executed. The code outside the `if` (the last `printl`) will always be executed.

Let's move on; we talked about the relational and logical operators. Logical operators are responsible for combining several relational expressions into one; there are three: `and`, `or` and `not`.

The `and` operator will make the expression to be true only when both its left part and right part are true. The `or` operator will cause the expression to be true when at least one of the left or right expressions is true. The `not` operator will reverse the truthfulness of the expression that follows it.

For example, `age > 8 and age < 12` would be true when the age is 9, 10 or 11, while `age = 9 or age = 10` would be true when age is 9 or 10.

As in algebra, the logical operators have a precedence. The `or` operator has the least precedence, `and` has a higher precedence and `not` has the highest; this means that any `or` will be considered after any `and` , and both of those will be considered only after any `not`. In this example:

```
a == 0 or a > 2 and  not a > 8
```

The expression is true if a is 0 (notice the double '=='), or if a is greater than 2 but not greater than 8. You can make things generally much easier to read adding parenthesises, like this:

```
(a == 0 or a > 2) and  not a > 8
```

This expression will held true if a is 0 or a is greater than 2. Additionally, in both cases the variable must be not greater than 8. Variables may also hold a truth value. In this example a is assigned the value of the expression opposite it:

```
b = -1
c = -2
a = ( b > 0 or c > 0)
/* some code here */
if a: printl( "a is true" )  // typing > a at the prompt would output 'false'
```

A variable is considered false if it holds 0, an empty string (like ""), an empty collection or the special value `nil` . It is true in every other case. Two base logic values are provided: `true` and `false` can be used as special values, one always being true and the other always evaluating to false; they are equal only to themselves. For example:

```
a = true
a == true: printl( "A is true" ) // same as if a: ... if
```

The value of an assignment always matches the final value of the variable that received the assignment. In other words:

```
a = 1
```

```
(a += 1) == 2: > "a is now 2"     // a was 1 but became 2 by the time it was evaluated if

a -= 2: > "a is now 0, so this check is 'false' and this line won't be printed"   if
```

Be sure not to write "=" instead of "==" when you want to check for a value! Another way to execute a piece of code based on a condition is the so-called fast-if operator. The definition is as follows:

```
<condition> ? <if true> [ : <if false>]
```

Actually, this operator (also known as the ternary operator or "`?:`") is directly borrowed from C. The value of the whole expression will be the expression right after the question mark if the condition is true, and the expression right after the colon if the condition is false. The latter may be missing; in that case, if the condition is false the whole expression will assume the nil value. Here are some examples:

```
var = is_process_done ? "Done" : "Still incomplete"  // assigns text to a variable

> is_process_done ? "Done" : "Still incomplete"        // outputs text

 if is_process_done ? first_test() : second_test()
        printl( "One of the two tests had been successful" )   // reports the a result
end
```

The switch statement

When a single value is needed to be checked against several different values, the `if` statement is a little clumsy. The switch statement checks a single value against several cases, providing a different set of statements to be executed in each of these conditions. The only limitation of switch is that the expressions it can examine must be strings or integers.

The prototype of a switch is as follows:

```
switch  expression

  case  item [, item, .. item ]
    statements...

  case  item [, item, .. item ]
    statements...

   /* other cases */

  default
    statements...
end
```

With `switch` a case statement may present one or more items, each of them separated by commas. If any of them is equal to the switch expression, then that case's statements are executed. Items that can be used as case selectors are:

- Integer numbers
- String literals
- Integer intervals
- Variable names
- The nil literal

Switches are actually powerful statements that can handle in one single VM step a very complex set of operations. If the item resulting from the switch expression is a number, it is checked against the

integer cases and intervals. If it is a string, it is checked against all the string cases. These operations are actually performed as binary searches on the cases, so the selection of a case is quite fast. Ranges of integers to be checked can be declared with the `to` keyword as in this example:

```
switch expression
  ...
    case  1  to  13, 20, 30, 40, 50, 60, 70, 80, 90  to  100
      /* Special number formatting handling */
      ...
```

Case items can also be simple variable names. Since their value can't be known in advance, and the check will be performed by scanning all the given variables for one matching the value of the expression in the order they are declared.

An optional `default` clause may be present and will be executed if none of the cases can be matched against the switch expression. This is an example:

```
print( "Enter your age: >" )
age = int( input() )
switch age
    case 1 to 5
      printl( "You are too young to program. Maybe." )

    case 6, 7, 8
      printl( "You may already be a great Falcon programmer." )

    case 9, 10
      printl( "You are ready to become a programmer." )

    default
      printl( "What are you waiting for? Start programming NOW" )
end
```

As mentioned before, the switch statement may also be used to check against strings:

```
switch month
    case  nil
      > "Undefined"

    case "Jan", "Feb", "Mar"(P43)
      > "Winter"

    case "Apr", "May", "Jun"
      > "Spring"

    case "Jul", "Aug", "Sep"
      > "Summer"

    default
      > "Autumn"
end
```

And symbols (that is, variable names) can be used as well.

```
switch func
    case print
      printl( "It was a print !!!" )

    case printl
      printl( "It was a printl !!!" )

    case MyObject
      printl( "The value of func is the same of MyObject" )
end
```

The values of symbols are determined at runtime. That is, in the above example, we may assign everything to `MyObject`, including `printl`. If this happens, the first matching case gets selected,

as if a set of `if/else` statements were used to determine the value of `func` .

However, since these checks are performed by the virtual machine in just one loop, the `switch` statement is much more efficient than a set of `if/else statements` (and it's more elegant), so when there's the need to check a variable against values that may be held in other variables, or against constant integer or string values, it's always better to use the `switch` statement.

As switch is a multi-line statement by nature, and since it should hold at least a "case" statement, it cannot be abbreviated on a single line with the colon (":"); however, each case can be placed on a single line if it improves the look of the code.

The select statement

The `select` statement is a `switch` that considers the type of the selected variable, rather than considering its value.

```
select variable
    case  TypeSpec
     ...statements...

    case  TypeSpec1 , TypeSpec2 , ..., TypeSpecN
     ...statements...

    default
     ...statements...
end
```

A `TypeSpec` can either be one of the pre-defined variable types or a symbol defined in the program. The symbol may be a variable, or it may be one of the things that has not yet been introduced: it may be a function, a class or an object instance.

Predefined types are the following:

- NilType
- BooleanType
- IntegerType
- NumericType
- RangeType
- MemBufType
- FunctionType
- StringType
- ArrayType
- DictionaryType
- ObjectType
- ClassType
- MethodType
- ClassMethodType
- OpaqueType

Predefined types get priority over class and instance cases, so a case such as "ObjectType" will prevent any branch on classes and objects ever to be considered. Case checking for symbols is performed in the order they are declared.

Even if obj is derived from Beta, as the check on derivation from Alfa comes first, that branch will be executed. In case of select statements testing related classes, the topmost siblings should be listed

first.

As with the `switch` statement, `select` case and default statements can be shortcut with the colon. The following is an example:

```
select param

    case IntegerType, NumericType
      return "number"

    case StringType: return "string"

    case Test2: return "class test2"
    case Test: return "class test"
    case Instance: return "instance obj"
    default : return "something else"
end
```

As with switch statements, symbolic case branches can actually be any kind of variable. Being dynamic, their contents may change at runtime.

The while statement

The while statement is the most important loop control statement. A loop is a set of zero or more statements that can be repeated more than once; usually there is a condition that causes the loop to be interrupted at some point.

The prototype of the while statement is:

```
while expression
    statements...
    [break]
    statements...
    [continue]
    statements...
end
```

While the expression is true, that is, each time the expression is found true, the statements are repeated. When the "end" keyword is reached the `while` expression is evaluated again, and if it's still true, the statements are repeated. The `while` statement may contain two special statements called `break` and `continue` . The break statement will cause the loop to be immediately terminated, and the Falcon VM will execute the first statement right after the end keyword. The continue statement does the opposite: it brings control immediately back to the while condition evaluation. If the condition is still true, the loop is repeated from start, and if it's now false the loop is terminated. (P6)Look at this example:

```
>> "Enter your age: > "

age = int( input() )
count = 0
while count < age

    if count == 18
      printl( "Congratulations, you may be able to vote!" )
      count += 1
      continue
    end

    if count == 23
      printl( "Ok, I'm bored with this." )
      break
    end

   count += 1
```

```
   > "Happy belated birthday for your ", count, "."

end
```

Please notice that we have used ">>" here to print the first line without appending a newline after it, and that the last ">" is followed by a list of values. In fact, the ">>" and ">" commands work respectively like `print()` and `printl()`, and, just like those two functions, they have the ability to accept more than one parameter. From now on we'll use mainly `print()/printl()` calls in the examples. This example will print some compliments for seventeen times; at the eighteenth it will change behavior. Notice that we must increment the counter before using the continue statement, because without this the while expression will be true again, and the number won't change. If you want to experiment with your first endless loop, remove that instruction. You can then stop the program by pressing CTRL+C. Then, when it comes to twenty three, an `if` statement causing immediate interruption of the loop will be executed. Notice that it's always possible to create endless loops that have an internal break sequence:

```
count = 0
while true

    if count > 100
      break
    end

   // do something here

   count += 1
end
```

But this is less efficient and somewhat confusing for the human reader. Sometimes you'll want it anyway, but you must have a good reason for having a `while` loop work this way. A `while` statement can be abbreviated with the trailing colon if the loop is composed by only a statement. One example might be:

```
 while var < 10 : var = calc( var )
```

hoping that `calc(var)` will somehow increase var to more than 10, sooner or later.

The loop statement

The `loop` works similarly to the `while` statement; it loops up to the moment where an *end condition* is met. The end condition is given after the `end` keyword, and is evaluated after the loop is performed at least once. Use of the `end` condition is optional; if not given, the loop goes on forever (or until a `break` statement is given inside it).

In example, to repeat a code section until a counter becomes 100:

```
count = 0
loop
   > ++ count
end count == 100
```

which is equivalent to:

```
count = 0
loop
   > ++ count

   if count == 100
      break
```

```
   end
end
```

As with all the other block statements, the `loop` statement may be shortened with the "`:`" colon.

More on lines and statements

It is possible that an expression will not fit on a text line. There are ways to avoid it, i.e. putting the partial expression values inside short name variables like so:

```
a = a_very_long_name and another_long_name
b = a_very_very_long_name and AnotherNameWithCapitals
 if a or b
//...
end
```

This usually improves readability of the scripts; but it costs memory, and sometimes it is just not possible to split expressions into different lines.

It's possible to split a statement on more than one line by putting a backslash ("\") at the end of it:

```
if a_very_long_name     and    another_long_name     or  \
        a_very_very_long_name and AnotherNameWithCapitals

   /* some statements here */

end
```

When you do so, it's a good habit to indent the exceeding part of the statement many times, so that it can be seen that it belongs to the upper statement, and to separate the block statements with at least a blank line.

When evaluating very long expressions, or when passing many long parameters to a function, having to use the backslash is a pain. So, in scripts, Falcon keeps track of the open parenthesis, and allows splitting the statement up to when the parenthesis is closed. In lists of elements, the comma may be followed by a new line without the need for a backslash. Look at this example:

```
printl( "This is a very long function call ",
        "which spawns on several lines ",
        "and includes long math such as: ",
        (alpha_beta + gamma) * 15 - 2 +
           psi + fi )

printl( "This is shorter" )
```

As it is not possible to put any statement inside a parenthesis, you don't have to worry about the fact that you may accidentally forget to close one; the compiler will raise an error when it finds a statement that looks as it were inside an expression, and sooner or later you'll have to put a statement somewhere.

This "auto-statement continuation" is useful while defining arrays or dictionaries, that we'll see later, because it allows declaring one item on each line without having to add an escape character at the end of the line.

Sometimes it's a good idea to split a statement onto two lines; sometimes you'll want the opposite. There are some code slices that are better read and managed if they are kept tiny, in a handful of lines. This can be achieved by separating different statements with the semicolon sign (";") and placing them on the same line:

```
print( "Expression is " );  a += 2; printl( a )
```

: By using the semicolon, you are putting different statements on the same line. There's no way in which you can use the colon "short statement block" indicator to have more than one statement to be considered by that method. Using a shortened statement and a semicolon will just cause the second statement to be considered as if it were separated. The compiler won't warn about this fact, that may pass unobserved; for example:

```
a == 0: print( "Expression is " );  a += 2; printl( a ) if
```

This line of code may look like as if it were executed only if a is 0. Indeed, just the first print is executed in that case; but the other two statements are executed regardless, and that may not be a desirable thing. A similar effect may be achieved instead with:

```
 if a == 0 ; print( "Expression is " );  a += 2; printl( a ); end
```

Notice the semicolon instead of the colon after the `if` . This is just an `if` statement, a block of three statements and an end written on the same line. As the compiler will complain if the end is missing (not immediately, just when it will find some incongruity in the control flow, but at least it will warn about it) this code is safer than the one above that may look to be doing one thing and actually do something else. You have to be careful when you put more than one statement on a line, and so, when you use shortened statements with the ":" colon operator the general rule is to avoid the practice unless you are sure that 1) you can't get confused and 2) statements look better on a single line.

Constant declarations

Sometimes it's useful to create a set of constants that may be used as symbols. This is useful to parameterize scripts at compile time, so that they can, for example, behave differently on different platforms, or just to associate a number with a symbolic meaning. The `const` keyword defines a constant and has this definition:

```
const name = immediate_value
```

An immediate value can be a number, a literal string, the `nil` keyword or another already defined constant. Once a constant is defined, it can be used in the program as if it were a variable:

```
const loop_times = 3

for i in [1:loop_times]
   > "looping: ", i
end
```

Remember that `const` can only declare constants that will be visible in the module currently being compiled, and only from the point where they are declared onwards.

A more organic constant values declaration, which may also be made available in foreign modules, is the `enum` keyword. This keyword creates a list of correlated constants which may assume any value. If a value is not declared, the constants are given an integer value starting from zero. For example :

```
enum Seasons
   spring
   summer
   autumn
   winter
end

> Seasons.spring    // 0
```

```
> Seasons.winter    // 3
```

The numeric values given to these constants are generated by adding 1 to the previous numeric value (rounded down); so it's possible to alter the sequence and to insert strings like this:

```
enum Seasons
   spring = 1                  // 1
   summer                      // 2
   midsummer = "So hot..."     // "So hot..."
   endsummer                   // 3 (string skipped)
   autumn = 10.12              // 10.12
   winter                      // 11
end
```

As the `enum` keyword is meant for readability, it's suggested that this feature be used widely to set a starting point, or to mix strings, numeric and nil values coherently.

Enumerated variables are runtime constants. This means that the compiler won't complain if an assignment to an enumerated constant is found, but an error will be raised in case a script actually tries to change one of those values.

About the for/in loop

A last control structure is the for/in loop; as its main function is that to iterate over particular data types (strings, arrays, dictionaries, lists, ranges and so on) it is presented in the next chapter, after having introduced those data types in detail.

Basic Datatypes

Arrays

The most important basic data structure is the array. An array is a list of items, in which any element may be accessed by an index. Items can also be added, removed or changed.

Arrays are defined using the [] parenthesis. Each item is separated by the other by commas, and may be of any kind, including the result of expressions:

```
array = ["person", 1, 3.5, int( "123" ), var1 ]
```

Actually the square brackets are optional; if the list is short, the array may be declared without parenthesis:

```
array = "person", 1, 3.5, int( "123" ), var1
```

But when using this method it is not possible to spread the list on multiple lines without using the backslash. Compare:

```
array = [ "person",
          1,
          3.5,
          int( "123" ),
          var1
        ]

array = "person", \
        1, \
        3.5, \
```

```
      int( "123" ), \
      var1
```

These two statements do the same thing, but the first is less confusing.

A list may be immediately assigned to a literal list of symbols to "expand it". This code:

```
a, b, c = 1, 2, 3
```

Will cause 1 to be stored in a, 2 to be stored in b, and 3 in c. More interestingly:

```
array = 1, 2, 3     // a regular array declaration
/* Some code here */
a, b, c = array     // the array's contents get copied to single variables
```

This will accomplish the same thing, but having the items packed in one variable makes it easier to carry them around. For example, you may return multiple values from a function and unpack them into a set of target variables. If the size of the list is different from the target set, the compiler (or the VM if the compiler cannot see this at compile time) will raise an error.

An item may be accessed by the [] operator. Each item i numbered from 0 to the size of the array -1. For example, you may traverse an array with a for loop like this:

```
var1 = "something"
array = [ "person", 1, 3.5, int( "123" ), var1 ]
i = 0
while i < len( array )
   printl( "Element ", i,": ", array[i] )
   i++
end
```

The function `len()` will return the number of items in the array. Array items also provide a method called `len` which allows extraction of the length of the array through the "dot" object access operator; the above line may be rewritten as:

```
while i < array.len(): > "Element ", i,": ", array[i++]
```

A single item in an array may be modified the same way by assigning something else to it:

```
array[3] = 10
```

An element of an array may be any kind of Falcon item, including arrays. So it is perfectly legal to nest arrays like this:

```
array = [ [1,2], [2,3], [3,4] ]
```

Then, array[0] will contain the array [1, 2]; array[0][0] will contain the number 1 from the [1,2] array.

Array indexes can be negative; a negative index means "distance from the end", -1 being the last element, -2 the element before the last and so on. So

```
array[0] == array[ - len(array) ]
```

always holds true (with a list that has at least one element).

Trying to access an item outside the array boundaries will cause a runtime error; this runtime error can be prevented by preventively checking the array size and the type of the expression we are using to access the array, or it can be intercepted as we'll see later.

It is possible to access more than one item at a time; a particular expression called "range" can be

used to access arrays and extract or alter parts of them. A range is defined as a pair of integers so that R=[n : m] means "all items from n to m-1". The higher index is exclusive, that is, excludes the element before the specified index, for a reason that will be clear below. The high end of the range may be open, having the meaning "up to the end of the array". As the beginning of the array is always 0, an open range starting from zero will include all elements of the array (and possibly none). The following shows how a range is used:

```
var1 = "something"
list = [ "person", 1, 3.5, int( "123" ), var1 ]
list1 = list[2:4]   // 3.5, int( "123" )
list2 = list[2:]    // 3.5, int( "123" ), "something"
list3 = list[0:3]   // "person", 1, 3.5
list4 = list[0:]    // "person", 1, 3.5, int( "123" ), "something"
list5 = list[:]     // "person", 1, 3.5, int( "123" ), "something"
```

A range can contain negative indexes. Negative indexes means "distance from end", -1 being the last item:

```
list1 = list[-2:-1]  // the element before the last
list2 = list[-4:]    // the last 4 elements.
list3 = list[-1:]    // the last element.
```

Finally, an array can have a range with the first number being greater than the last one; in this special case the last index is inclusive (note that the last element is counted in the resulting list). This produces a reverse sequence:

```
list1 = list[3:0]   // the first 4 elements in reverse order
list2 = list[4:2]   // elements 4, 3 and 2 in this order
list3 = list[-1:4]  // from the last element to the 4th
list4 = list[-1:0]  // the whole array reversed.
```

Don't be confused about the fact that negative numbers are "usually" smaller than positive ones. A negative array index means the end of the array -x, which may be smaller or greater than a positive index. In an array with 10 elements, the element -4 is greater than the 4 (10-4 = 6), while in an array of 6 elements, -4 is smaller than 4 (6-4 = 2).

Ranges can be independently assigned to a variable and then used as indexes at a later time:

```
if a < 5
   rng = [a:5]
else
   rng = [5:a]
end
array1 = array[rng]
```

Of course, both the array indexes and the range indexes may be a result from any kind of expression, provided that expression evaluates to a number.

To access the beginning or the end of a range, you may use the array accessors; the index 0 is the first element, and the index 1 (or -1) is the last. If the range is open, the value of the last element will be nil.

```
rng = [1:5]
printl( "Start: ", rng[0], ";   End: ", rng[1] )
rng = [1:]
printl( "Will print nil: ", rng[1] )
It is possible to assign items to array ranges:
b, c = 2, 3
list[0:2] = b       // removes items 0 and 1, and adds b in their place
list[1:1] = c       // inserts c at position 1.
list[1] = []        // puts an empty array in place of element 1
list[1:2] = []      // removes item 1, reducing the array size.
```

As the last two rows of this example demonstrates, assigning a list into an array range causes all the original items to be changed with the new list ones; they may be less, more or the same than the original ones. In particular, assigning an empty list to a range causes the destruction of all the items in the range without replacing them.

The fact that the end index is not inclusive allows for item insertion when using a range that does not include any items: [0:0] mean "inserts some item at place 0", while [0:1] indicates exactly the first item.

To extend a list it is possible to use the plus operator "+" or the self assignment operator:

```
a = [ 1, 2 ]
b = [ 3, 4 ]
c = a + b          // c = [1, 2, 3, 4]
c += b             // c = [1, 2, 3, 4, 3, 4]
c += "data"        // c = [1, 2, 3, 4, 3, 4, "data"]
a += []            // a = [1, 2, [] ]
a[2] += ["data"]   // a = [1, 2, ["data"] ]
```

To remove selectively elements from an array, it is possible to use the "-" (minus) operator. Nothing is done if trying to remove an item that is not contained in the array:

```
a = [ 1, 2, 3, 4, "alpha", "beta" ]
b = a - 2                          // b = [ 1, 3, 4, "alpha", "beta" ]
c = a - [ 1, "alpha" ]             // c = [ 2, 3, 4, "beta" ]
c -= 2                             // c = [ 3, 4, "beta" ]
a -= c                             // a = [ 1, 2, "alpha"]
a -= "no item"                     //  a is unchanged; no effect
```

Array manipulation functions

Falcon provides a set of powerful functions that complete the support for arrays. A preallocated buffer containing all nil elements can be created with the arrayBuffer function:

```
arr = arrayBuffer(4)
arr[0] = 0
arr[1] = 1
arr[2] = 2
arr[3] = 3
inspect( arr )
```

This prevents unneeded resizing of the array when its dimension is known in advance.

To access the first or last element of an array, for example, in loops, `arrayHead` and `arrayTail` functions can be used. They retrieve and then remove the first or last element of the array. For example, to pop the last element of an array:

```
arr = [ "a", "b", "c", "d" ]
while arr.len() > 0
   > "Popping from back... ", arrayTail( arr )
end
```

It is possible to remove an arbitrary element with the `arrayRemove` function, which must be given the array to work on and the index (eventually negative to count from the end). More flexible are the `arrayDel` and `arrayDelAll` functions. The former removes the first element matching a given value; the latter removes all the matching elements:

```
a = [ 1, 2, "alpha", 4, "alpha", "beta" ]
arrayDelAll( a, "alpha" )
inspect( a )   // "alpha" has been removed
```

The `arrayFilter` function is still more flexible and allows the performance of a bit of functional programming over arrays (note that `arrayFilter` is similar to the `filter()` functional construct that we'll see later on). This function calls a given function providing it with one element at a time; if the function returns true, the given element is added to a final array, otherwise it is skipped. We haven't introduced the functions yet, so just take the following example as-is:

```
function passEven( item )
   return item.type() == IntegerType and item % 2 == 0
end

array = [1, 2, 3, 4, "string", 5, 6]
inspect( arrayFilter( array, passEven ) )
```

To search for an element in an array, `arrayFind` and `arrayScan` functions can be used. The `arrayFind` functions returns the index in the array of the first element matching the second parameter, while `arrayScan` works like `arrayFilter` and returns the indexes at which the called function returned true. For example:

```
a = [ 1, 2, "alpha", 4, "alpha", "beta" ]
> "First alpha is found at... ", arrayFind( a, "alpha" )
```

Be sure to read the Array function section in the Function Reference for more details on the topic.

Comma-less arrays

When there are very long sequences of items, or when functional programming is involved, using a comma to separate tokens can be a bit clumsy and error prone.

Commas offer a certain protection against simple writing errors, but once you gain a bit of confidence with the language, it is easier to use the "dot-square" array declarator. The following declarations are equivalent:

```
a1 = [1, 2, 3, 'a', 'b', var1 + var2, var3 * var4, [x,y,z]]
a2 = .[ 1 2 3 4 'a' 'b' var1 + var2 var3 * var4 .[x y z]]
```

When using this second notation, it is important to be careful about parenthesis as they may appear to be function calls, strings (they may get merged, as we'll see in the next chapter), sub-arrays (they may be interpreted as the index accessor of the previous item) and so on, but when programming in a functional context, where function calls and in-line expression evaluations are rare if not forbidden, this second notation may feel more natural.

The arrays declared with dot-square notation may contain commas if this is necessary to distinguish different elements; in this case, consider putting the comma at the immediate left of the element that they are meant to separate. For example, here's an array in which we need a range after a symbol:

```
array = .[ somesym  ,[1:2] ]
```

In this case without the comma separating the two, the range would be applied to the preceding symbol.

Strings

Other than a basic type, strings can be considered a basic data structure as they can be accessed exactly like arrays that only have characters as items. Characters are treated as single element strings (they are just a string of length 1). It is possible to assign a new string to any element of an

older one. Here is an example of the string functionality:

```
string = "Hello world"

/* Access test */
i = 0
while i <= len( string ) - 1
   >> string[i], ","  // H,e,l,l,o, ,w,o,r,l,
end
> string[-1]          // d

/* Range access tests */
printl( string[0:5] )       // Hello
printl( string[6:] )        // world
printl( string[-1:6] )     // dlrow
printl( string[-2:] )       // ld
printl( string[-1:0] )     // dlrow olleH

/* Range assignment tests */
string[5:6] = " - "
printl( string )            // Hello - world
string[5:8] = " "
printl( string )            // Hello world

/* Concatenation tests */
string = string[0:6] + "old" + string[5:]
printl( string )            // Hello old world
string[0:5] = "Goodbye"
string[8:] = "cruel" + string[7:]
printl( string )            // Goodbye cruel old world

/* end */
```

Assigning a string to a single character of another string will cause that character to be changed with the first character from the other string:

```
string = "Hello world"
string[5] = "-xxxx"    // "Hello-world", the x characters are not used
```

Multiline strings

Strings can span multiple lines; starting the string with a single/double quote followed directly by an End Of Line (EOL) will cause the string to span on multiple lines, until another quote character is found.

```
longString = "
     Aye, matey, this is a very long string.
     Let me tell you about my remembering
     of when men were men, women were women,
     and life was so great."

printl( longString )
```

You'll notice that the spaces and tabs in front of each line are not inserted in the final string; this is to allow you to create wide strings respecting the indentation of the original block. To insert a newline, the literal `\n` can be used. It is also possible to use the literal multiline string (see below).

To perserve the whole formating (to include newlines) in a string declaration one can use single quotes. Here is a quick example.

```
str = '
ABC
123
'

dstr = "
ABC
```

```
123
"

iStr = '
国際ストリング
国際ストリング
'

iDStr = "
国際ストリング
国際ストリング
"

> @ "1 $str"
> @ "2 $dstr"
> @ "3 $iStr"
> @ "4 $iDStr"
```

The above will output:

```
1 ABC
123

2 ABC 123
3 国際ストリング
国際ストリング

4 国際ストリング  国際ストリング
```

A finer control can achieved through explicit string concatenation, using the + operator (that can be placed also at the end of a line to concatenate with the following string):

```
longString = "Aye, matey, this is a very long string.\n" +
             "  Let me tell you about my remembering\n" +
             "      of when men were men, women were women,\n" +
             "            and life was so great."
printl( longString )
```

You will have a long string on the console.

Falcon strings support escape characters in C-like style: a backslash introduces a special character of some sort. Suppose you want to format the above text so that every line goes one after another, with a little indentation so that it is known as a "citation".

```
longString = "\t Aye, matey, this is a very long string.\n" +
              "\t   Let me tell you about my remembering\n" +
              "\t   of when men were men, women were women,\n" +
              "\t   and life was so great."

printl( longString )
```

The `\n` sequence tells Falcon to skip to the next line, while the `\t` instructs it to add a "tab" character, a special sequence of (usually) eight spaces.

Other escape sequences are the \", \\, \b and \r. The sequence \" will put a quote in the string, so that it is possible to also print quotes; for example:

```
printl( "This is a \"quoted\" string." )
```

The "\\" sequence allows the insertion of a literal backslash in the string, for example:

```
myfile = "C:\\mydir\\file.txt"
```

will be expanded into C:\mydir\file.txt

The `\r` escape sequence is used to make the output restart from the beginning of the current line. It's a very rudimentary way to print some changing text without scrolling the text all over the screen, but is commonly used for effects like debug counters or console based progress indicators. Try this:

```
i = 0
while i < 100000
   print( "I is now: ", i++ , "\r" )
end
```

Similarly, the `\b` escape causes the output to go back exactly one character.

```
print( "I is now: " )
i = 0
while i < 100000
   print( i )
   if i < 10
        print( "\b" )
   elif i < 100
        print( "\b\b" )
   elif i < 1000
        print( "\b\b\b" )
   elif i < 10000
        print( "\b\b\b\b" )
   else
        print( "\b\b\b\b\b" )
   end
   i++
end
printl()
```

International strings

Falcon strings can contain any Unicode character. The Falcon compiler can input source files written in various encodings. UTF-8 and UTF-16 and ISO8859-1 (also known as Latin-1) are the most common; Unicode characters can also be inserted directly into a string via escapes. For example, it is possible to write the following statement:

```
string = "国際ストリング"
printl( string )
```

The `printl` function will write the contents of the string on the standard Virtual Machine output stream. The final outcome will depend on the output encoding. The Falcon command line sets the output stream to be a text stream having the encoding detected on the machine as output encoding. If the output encoder is not able to render the characters they will be translated into "?". Another method to input Unicode characters is to use numeric escapes. Falcon parses two kinds of numeric escapes: "\0" followed by an octal number and "\x" followed by an hexadecimal number. For example:

```
string = "Alpha, beta, gamma: \x03b1, \x03B2, \x03b3"
printl( string )
```

The case of the hexadecimal character is not relevant.

Finally, when assigning an integer number between 0 and 2^32 (that is, the maximum allowed by the Unicode standard) to a string portion via the array accessor operator (square brackets), the given portion will be changed into the specified Unicode character.

```
string = "Beta: "
string[5] = 0x3B2
printl( string )  // will print Beta:β
```

Accessing the nth character with the square brackets operator will cause a single character string to be produced. However, it is possible to query the Unicode value of the nth character with the bracket-star operator using the star square operator ([*]):

```
string = "Beta:β"
i = 0
while i < string.len()
    > string[i], "=", string[* i++]
end
```

This code will print each character in the string along with its Unicode ID in decimal format. If you need to internationalize your program, you may want to examine the Program Internationalization section.

String replication

It is possible to replicate a string a certain number of times using the `*` (star) operator. For example:

```
  sep = "-~*~-" * 12
  > sep
  > " "*25 + "Hello there!"
  > sep
```

Notice the expression `" "*25 + "Hello there!"`, concatenating the result of the first string expansion to the last part of the string.

String-to-number concatenation

Adding an item to a string causes the item to be converted to string and then concatenated. For example, adding 100 to "value" ...

```
string = "Value: " + 100
> string                          // prints "Value: 100"
```

In the special case of numbers, it is possible to add a character by its *unicode* value to a string through the `%` (modulo) operator. For example, to add an "A" character, whose unicode value is 65, it is possible to do:

```
string = "Value: " % 64
> string                          // prints "Value: A"
string %= 0x3B2
> string                          // "Value: Aβ"
```

The `/` (slash) operator modifies the value of the last character in the string, adding to its UNICODE value the value you provide. For example, to get 'd' from 'a' and the other way around:

```
d_letter = "a" / 3           // chr( ord('a') + 3) == 'd'
a_letter = d_letter / -3     // chr( ord('d') - 3) == 'a'
> a_letter, ", ", d_letter
```

String polymorphism

In Falcon, to store and handle efficiently strings, strings are built on a buffer in which each character occupies a fixed space. The size of each character is determined by the size in bytes needed by the widest character to be stored. For Latin letters, and for all the Unicode characters whose code is less than 256, only one byte is needed. For the vast majority of currently used alphabets, including Chinese, Japanese, Arabic, Hebrew, Hindi and so on, two bytes are required.

For unusual symbols like musical notation characters four bytes are needed. In this example:

```
string = "Beta: "
string[5] = 0x3B2
printl( string )  // will print "Beta:β"
```

the string variable was initially holding a string in which each character could have been represented with one byte.

The string was occupying exactly six bytes in memory. When we added β the character size requirement changed. The string has been copied into a wider space. Now, twelve characters are needed as β Unicode value is 946 and two bytes are needed to represent it.

When reading raw data from a file or a stream (i.e. a network stream), the incoming data is always stored byte per byte in a Falcon string. In this way binary files can be manipulated efficiently; the string can be seen just as a vector of bytes as using the [*] operator gives access to the nth byte value. This allows for extremely efficient binary data manipulation.

However, those strings are not special. They are just loaded by inserting 0-255 character values into each memory slot, which is declared to be 1 byte long. Inserting a character requiring more space will create a copy of each byte in the string in a wider memory area.

Files and streams can be directly loaded using transcoders. With transcoder usage, loaded strings may contain any character the transcoder is able to recognize and decode.

Strings can be saved to files by both just considering their binary content or by filtering them through a transcoder. In the case that a transcoded stream is used, the output file will be a binary file representing the characters held in the string as per the encoding rules.

Although this mixed string valence, that uses fully internationalized multi-byte character sequences and binary byte buffers, could be confusing at first, it allows for flexible and extremely efficient manipulation of binary data and string characters depending on the need.

It is possible to know the number of bytes occupied by every character in a string through the `String.charSize` method of each string; the same method allows to change the character size at any moment. See the following example:

```
str = "greek: αβγ"
> str.charSize()      // prints 2
str.charSize( 1 )      // squeeze the characters
> str                        // "greek: " + some garbage
```

This may be useful to prepare a string to receive international characters at a moment's notice, avoiding paying the cost for character size conversion. For example, suppose you're reading a text file in which you expect to find some international characters at some point. By configuring the size of the accumulator string ahead of time you prevent the overhead of determining character byte size giving you a constant insertion time for each operation:

```
str = ""
str.charSize( 2 )
file = ...

while not file.eof()
   str += file.read( 512 )
end
```

Valid values for String.charSize() are 1, 2 and 4

Literal Strings

Strings can be also declared with single quotes, like this:

```
str = 'this is a string'
```

The difference with respect to the double quote is that literal strings do not support any escape sequence. If you need to insert a single quote in a literal string, use the special sequence `''` (two single quotes one after another), like in the following example:

```
> 'Hello ''quoted'' world!'      // Will print "Hello 'quoted' world"
```

Parsing of literal strings is not particularly faster or more efficient than parsing of standard strings; they have been introduced mainly to allow short strings with backslashes to be more readable. For example, they are useful with Regular expressions where backslashes already have a meaning.

When used as multiline strings, single quoted strings will include all the EOL and blanks present in the source. For example:

```
multi = '
   Three spaces before me...
    And four here, and on a new line
     and now five on the final line.'

printl( multi )
```

String expansion operator

Many scripting languages have a means to "expand" strings with inline variables. Falcon is no exception, and actually it adds an important functionality to currently known and used string expansion constructs: inline format specifications. This combination allows for an extreme precise and powerful "pretty print" construct which we are going to show now in detail.

Strings containing a "$" followed by a variable can be expanded using the unary operator "@". For example:

```
value = 1000
printl( @ "Value is $value" )
```

This will print "Value is 1000". Of course, the string can be a variable, or even composed of many parts. For example:

```
value = 1000
chr = "$"
string = "Value is " + chr +"value"
> "Expanding ", string, " into ", @ string
```

The variable after the "$" sign is actually interpreted as an "accessible" variable; this means that it may have an array accessor like this:

```
array = [ 100, 200, 300 ]
printl( @ "Array is $array[0], $array[1], $array[2]"  )
Actually, everything parsed inside an accessor will be expanded. For example:
array = [ 100, 200, 300 ]
value = 2
printl( @ "The selected value is $array[ value ]"  )
```

The object member "dot" accessor can also be used and interleaved with the array accessor; but we'll see this in the character dedicated to objects. For now, just remember that a "." cannot

immediately follow a "$" symbol, or it will be interpreted as if a certain property of an object were to be searched.

```
printl( @ "The selected value is $(array[ value ])."  )
```

In this way the parser will understand that the "." after the array[value] symbol is not meant to be a part of the symbol itself. A string literal may be used as dictionary accessor in an expanded string either by using single quotes (') or escaping double quotes, but always inside parenthesis, as in this example:

```
dict = [ "a" => 1, "b" => 2]
> @ "A is $(dict['a']), and B is $(dict[\"b\"])"
```

To specify how to format a certain variable, use the ":" colon after the inlined symbol name and use a format string. A format string is a sequence of commands used to define how the expansion should be performed; a complete exposition is beyond the scope of this guide (the full reference is in the function reference manual, in the "Format" class chapter), but we'll describe a minimum set of commands here to explain basic usage:

- A plain number indicates "the size of the field"; that is, how many characters with which the output should be wrapped.
- the 'r' letter forces alignment to the right.
- A dot followed by a plain number indicates the number of decimals.

For example, to print an account book with 3 decimal precision, do the following:

```
data = [ 'a' => 1.32, 'b2' => 45.15, 'k69' => 12.4 ]

for id, value in data
   printl( @ "Account number $(id:3):$(value:8.3r)" )
end
```

The result is:

```
Account number a  :   1.320
Account number b2 :  45.150
Account number k69:  12.400
```

As it can be seen, the normal (left) padding was applied to the ID while the right padding and fixed decimal count was applied to the value. Formats can be also applied to strings and even to objects, as in this example:

```
data = [ "brown", "smith", "o'neill", "yellow" ]
i = 0
while i < data.len()
   value = data[i++]
   printl( @ "Agents in matrix:$(value:10r)" )
end
```

The result is:

```
Agents in matrix:     brown
Agents in matrix:     smith
Agents in matrix:   o'neill
Agents in matrix:    yellow
```

The sequence "$$" is expanded as "$". This makes possible to have iterative string expansions like the following:

```
value = 1000
str = @ "$$$value"
```

```
printl( str )

Or more compactly:
value = 1000
str = "$$$value"
> @ str
```

In case of a parsing error, or if a variable is not present in the VM (i.e. not declared in the module and not explicitly imported), or if an invalid format is given, an error will be raised that can be managed by the calling script. We'll see more about error raising and management later on.

String manipulation functions

Falcon provides functions meant to operate on strings and make string management easier. Classic functions such as trim (elimination of front/rear blank characters), uppercase/lowercase transformations, split and join, substrings and so on are provided. For example, the following code will split "Hello world" and work on each side:

```
h, w = strSplit( "Hello world", " " )
> "First letter of first part: ", strFront( h, 1 )
> "Last letter of the second part: ", strBack( w, 1 )
> "World uppercased: ", strUpper( w )
```

Several interesting functions are `strReplicate` that builds a "stub" sequence repeating a string, and `strBuffer`, which creates a pre-allocated empty string. A string allocated with `strBuffer` can then be used as buffer for memory based operations such as iterative reading of data blocks from binary files.

For more details on Falcon's support of strings, read the *String functions* section in the Function Reference.

Dictionaries

The most flexible basic structure is the Dictionary. A dictionary looks like an array that may have any object as its index. Most notably, the dictionary index may be a string. More formally, a dictionary is defined as a set of pairs, of which the first element is called key and the second value. It is possible to find a value in a dictionary by knowing its key. Dictionaries are defined using the arrow operator (=>) that couples a key with its value. Here is a minimal example:

```
dict = [ => ]         // creates an empty dictionary
dict = [ "a" => 123, "b" => "onetwothree" ]
printl( dict["a"] ,":", dict["b"] )     // 123:onetwothree
```

Of course, the keys and values can be expressions, resulting in both in dictionary definition and in dictionary access:

```
a = "one"
b = "two"
dict = [ a + b => 12, a => 1, b => 2 ]
printl( dict[ "onetwo" ] )              // 12
printl( dict[ a + b ] )                 // 12 again
```

Dictionaries do not support ranges. To extend a dictionary, it is possible to just name an nonexistent key as its index; if the element is an already existing key, the value associated with that key is changed. If it's a nonexistent key the pair is added:

```
dict = [ "one" => 1 ]
```

```
dict[ "two" ] = 2
dict[ "three" ] = 3
// dict is now [ "one" => 1, "two" => 2, "three" => 3 ]
```

It is also possible to "sum" two dictionaries; the resulting dictionary is a copy of the first addend, with the items of the second added being inserted over the first one. This means that in case of intersection in the key space, the value of the second addend will be used:

```
dict = [ "one" => 1, "two" => 2 ] + [ "three" => 3, "four" => 4 ]
// dict is now [ "one" => 1, "two" => 2, "three" => 3, "four" => 4 ]

dict = [ "one" => 1, "two" => 2 ] + [ "two" => "new value", "three" => 3 ]
// dict is now [ "one" => 1, "two" => "new value", "three" => 3 ]

dict += [ "two" => -2, "four" => 4 ]
// dict is now [ "one" => 1, "two" => -2, "three" => 3, "four" => 4 ]
```

On the other hand, accessing an nonexistent key will raise an error, like trying to access an array out its bounds:

```
dict = [ "one" => 1 ]
printl( dict[ "two" ] )    // raises an error
```

To selectively remove elements from a dictionary, it is possible to use the "-" (minus) operator. Nothing is done if trying to remove an item that is not contained in the dictionary:

```
a = [ 1=>'1', 2=>'2', "alpha"=>0, "beta"=>1 ]
b = a - 2              // b = [ 1=>'1', "alpha"=>0, "beta"=>1 ]
c = a - [ 1, "alpha" ] // c = [ 2=>'2', "beta"=>1 ]
c -= 2                 // c = [ "beta"=>1 ]
a -= b                 // a = [ 2=>'2' ]
a -= "no item"         // a is unchanged; no effect
```

Dictionary support functions

Falcon offers some functions that are highly important to complete the dictionary model. For example, the most direct and simple way to remove an item from a dictionary is to use the **dictRemove** function (or **remove** method):

```
a = [ 1=>'1', 2=>'2', "alpha"=>0, "beta"=>1 ]
dictRemove( a, "alpha" )
inspect( a )   // alpha is not in the dictionary anymore.

a.remove( "beta" )
inspect( a )   // and now, beta is gone too.
```

It is also possible to remove all the elements using the **dictClear** function or **clear** method). Other interesting functions are **dictKeys** and **dictValues** (again, with corresponding dictionary methods **keys** and **values**), which create a vector containing respectively all the keys and all the values in the dictionary.

Serious operations on dictionaries require the recording of a position and proceeding in some direction. For example, in the case it is necessary to retrieve all the values having a key which starts with the letter "N", it is necessary to get the position of the first element whose key starts with "N" and the scan forward in the dictionary until the key changes first letter or the end of the dictionary is reached.

To work on dictionaries like this, Falcon provides two functions called **dictFind** and **dictBest** (or **find** and **best** methods), which return an instance of a class called *Iterator*. We'll see more about iterators in the next sections.

Be sure to read the section called Dictionary functions in the function reference.

Lists

We have seen that arrays can be used to add or remove elements randomly from any position. However, this has a cost that grows geometrically as the size of an array grows. Insertion and removal in lists are more efficient, by far, when the number of elements grows beyond the size a simple script usually deals with. Switching from arrays to lists should be considered at about 100 items.

Contrary to strings, arrays and dictionaries, Lists are full-featured Falcon objects. They are a class, and when a list is created, it is an instance of the List class. Objects and classes are described in a further chapter, but Lists are treated here for completeness.

A list is declared by assigning the return value of the List constructor to a variable.

```
l = List( "a", "b", "c" )
Operations that can be performed on a list are inspection of the first and last element and
insertion and removal of an element at both sides.
Some examples below:
> "Elements in list: ", l.len()
> "First element: ", l.front()
> "Last element: ", l.back()

// inserting an element in front
l.pushFront( "newFront" )
> "New first element: ", l.front()

// Pushing an element at bottom
l.push( "newBack" )
> "New first element: ", l.back()

// Removing first and last element
l.popFront()
l.pop()
> "Element count now: ", l.len()
```

Lists also support iterator access; it's possible to traverse a list, insert or remove an element from a certain position through an iterator or using a **for/in** loop. We'll treat those arguments below.

The "in" operator

The in relational operator checks for an item to its left to be present in a sequence to its right. It never raises an error, even if the right operand is not a sequence; instead, it assumes the value of true (1) if the item is found or 0 (false) if the item is not found, or if the right element is not a sequence.

The in operator can check for substrings in strings, or for items in arrays, or for keys in dictionaries. This is an example:

```
print( "Enter your name > " )
name = input()

if "abba" in name
   printl( "Your name contains a famous pop group name" )
end

dict = [ "one" => 1 ]
if "one" in dict
   printl( "always true" )
end
```

There is also an unary operator notin working as not (x in name):

```
if "abba" notin name
   printl( "Your name does not contain a famous pop group name" )
end
```

The for/in loop

The for/in loop traverses a collection of items (an array, a dictionary, a list or other application/module specific collections), usually from the first item to the last one, and provides the user with a variable assuming the value of each element in turn. The loop can be interrupted at any point using the break statement, and it is possible to skip immediately to the next item with the continue statement. The value being currently processed can be changed with a special operator, called “dot assign”, and the continue dropping statement discards the currently processed item, continuing the processing loop from the next one.

The for/in loop can also be applied to strings, where it picks all the characters from the first to the last, and to ranges, to generate sequences of integer numbers. A special application of the for/in loop to ranges is the for/to loop, which follows a slightly different semantics.

Other than the main body, the for/in loop can contain three special blocks: `forfirst`, `forlast` and `formiddle` blocks can contain code that is respectively executed before the first item, after the last item, and after every item that is not the last (between items, essentially).

Loop control statements (namely break, continue and continue dropping) being declared in the main block will prevent `formiddle` and `forlast` blocks from being executed. If they are contained in the `forfirst` block, even the main block for the first item is skipped, as `forfirst` block is executed before the main block.

This is the formal declaration of the for/in block:

```
for variable[,variable...] in collection
   ...statements...
   [break | continue | continue dropping]
   ...statements...

   forfirst
       ... first time only statements ...
   end

   formiddle
       ... statements executed between element processing ...
   end

   forlast
       ... last time only statements ...
   end
end
```

The `forfirst`, `forlast` and formiddle blocks can be declared in any order or position; actually, the can even be interleaved with the main for/in block code; the code will just be separated and executed sequentially. As with any block, they can be abbreviated using the “:” colon shortcut.

This example will print “Contents of the array: Have a nice day!” on a single line.

```
array = [ "Have", "a", "nice", "day" ]

for element in array
   forfirst: print( "Content of the array: " )

   // this is the main for/in body
   print( element )
```

```
   formiddle: print( " " )
   forlast: printl( "!" )
end
```

Using `forfirst` and `forlast` blocks in the `for/in` loop will allow actions to take place only if the collection is not empty, exactly before the first element and after the last one. Using those blocks, there isn't the need of extra checks around the collection traversal loop. Also, the special blocks in the `for/in` loop are managed at VM level, and are considerably faster than using repeated checks in a normal loop.

An empty set, that is, an array or a dictionary with no elements, will cause the for/in loop to be skipped altogether. A nil value will be interpreted as an empty set, so the following:

```
array = nil

for element in array
   print( element, " " )
end
```

will just be just skipped. A for/in loop applied to a dictionary requires two variables to be used; the first one will receive the current key, and the second one will store the entry value:

```
dict = [ "Have" => 1 , "a" => 2, "nice" => 3, "day" => 4 ]

for key, value in dict
   printl( "Key: ", key, " Value: ", value )
end
```

This technique will also work with multidimensional arrays, provided that every element is an array of the same size:

```
matrix = [ [1, 2, 3], [3, 4, 5], [5, 6, 7] ]

for i1, i2, i3 in matrix
   printl( i1, ",", i2, ",", i3  )
end
```

The values which are retrieved in the for/in loop can also be changed on the fly. To do this, use the unary operator ".=" (called dot-assign), that changes the currently scanned item without altering the loop variable, like in this example:

```
array = [ 1, 2, 3 ,4, 5 ]

for elem in array
   .= 0           // sets all the array elements to zero...
   printl(elem)    // ... but prints the original items
end

for elem in array
   printl( elem )      // prints five zeros
end
```

In the case of a dictionary being traversed the function dot-assign operator will also change the current value of the dictionary. The current key of a dictionary cannot be changed.

To remove an item from a collection, use the continue dropping statement; for example, the following code will filter the source array so that only even numbers are left:

```
array = [ 1, 2, 3, 4, 5 ]

for elem in array
   if elem % 2 == 1
      continue dropping
```

```
   end
   printl( "We accepted the even number: ", elem )
end
```

As shown, the continue dropping statement will also skip the rest of the main for/in body, as well as `formiddle` and `forlast` blocks, if present.

For/in ranges

The range based for/in loop is an efficient way to generate increasing or decreasing values. The target variable of the for/in is filled each loop with an integer value.

If the range beginning is lower than the end, the index variable will be filled with values from the beginning, included, to the end, excluded. So:

```
for value in [1:10]
   printl( value )
end
```

will print a sequence between 1 and 9. Contrarily, if the beginning of the range is higher than the end, the variable will be filled with decreasing values, including the end limit. This resembles the way that ranges are used to extract substrings or subarrays.

If the range is open, or if it is empty ([n:n]), then the for/in loop is completely skipped.

The continue dropping statement is performed, but it is translated to a simple continue. The dot-assignment has no effect.

Ranges in for/in loop support steps; a third parameter may be specified in the range to indicate a stepping value; for example, the following loop shows the pair numbers between 0 and 10:

```
for value in [ 0: 11: 2 ]
   > value
end
```

If the step is zero, or if it's greater than zero when the loop would be descending or if it's less than zero when the loop would be ascending, the for/in statement is skipped. In case the direction of the loop is unknown because of variable parameters, the step can be used to ensure a certain processing order:

```
array = [ 1, 2, 3, 4, 5 ]
for index in [ start : len( array ) : 1 ]  // so if start too high, we'll just skip
   > array[ index ]
end
```

For/to loops

The for/to loop works as a for/in loop in ranges, but it includes the upper limit of the range. It is declared as:

```
for variable = lowerbound to upperbound [, step]
   // for/to body, same as for/in
end
```

For example, this will count 1 to 10 included, treating 1 and 10 a bit specially:

```
for i = 1 to 10
   forfirst: >> "Starting: "

   >> i
```

```
   formiddle: >> ", "
   forlast: > "."
end
```

The step clause works exactly as in for/in ranges, declaring the direction of a loop and eventually having the loop skipped if the direction is wrong. For example, this prints all the pair numbers between 1 and 10 included:

```
for i = 2 to 10, 2
   > i
end
```

For/in lists

Lists can be processed through a for/in loop exactly as arrays. Positional blocks will work as for any other type; continue dropping statement removes the current element, while the dot assign operator changes the value of the current element. For example:

```
list = List( "Have", "a", "nice", "day" )
for value in list
   forfirst: >> "The list is... "
   >> value
   formiddle: >> " "
   forlast: > "!"
end
```

Although lists can also be traversed with iterators, the for/in loop is completely VM driven, and thus it is more efficient; it also uses a simpler internal representation of the iterator, sparing memory.

Iterators have a small chapter on their own, as they are tightly bound with the Object Oriented Programming paradigm supported by Falcon; so, they will be presented after the chapter in which OOP support is described.

For/in generators

The for/in loop supports genetator functions since version Eagle (0.9.4). As we didn't introduce functions and other functional programming elements useful for this construct, we'll descend in details of this structure in the List Comprehension chapter.

Memory buffers

Memory buffers are “tables” of raw memory which can be directly manipulated through Falcon scripts. They are mainly meant to access binary streams or to represent memory mapped data (as images). They may be also used by modules and applications to pass a set of data in a very efficient way to the script, or the other way around, as each access to them refers to a small unsigned integer value in memory. Memory buffers can be sequences of numbers occupying one to four bytes (including three bytes, which is a quite common size for memory mapped images).

Memory buffers cannot grow nor shrink, and it is not possible to access a subrange of them. From a script standpoint, they are table of small integer values. Consider the following example:

```
memory = MemBuf( 5, 2 )  // creates a table of 5 elements, each 2 bytes long.
for value in [0:5]
   memory[ value ] = value * 256
end

inspect( memory )
```

The inspect function will show a set of two-bytes binary data inside the buffer:

```
MemBuf(5,2) [
0000 0100 0200 0300 0400 ]
```

Hexadecimal 0100 value equals 256, 0200 is 512 and so on.

Functions dealing with files may be given a string or a memory buffer to fill. In the second case, manipulation of binary data may be easier. Strings can be used to manipulate binary data too (as it is possible to access their content by the value of each character), but memory buffers are more fit for that task.

Bitwise operators

Dealing with binary data often requires checking, setting or resetting specific bits. Falcon support bitwise operations on integer data (memory buffer elements, string characters accessed by numeric value or integer items).

The bitwise and '&&', bitwise or '||', bitwise xor '^^' and bitwise not '~' operators allow the changing bits of integer values through binary math. And, or and xor operator are binary, while not operator is unary. For example,

```
value = 0x1 || 0x2   // or bits 0 and 1

// display binary:
> @"$(value:b)b = $(value:X)H = $value"

value = value && 0x1 // turns off bit 2
> @"$(value:b)b = $(value:X)H = $value"

value = ~value && 0xFFFF // Shows first 2 bytes of reversed value
> @"$(value:b)b = $(value:X)H = $value"

value = value ^^ 0x3  // turns off bit 2 and on bit 1
> @"$(value:b)b = $(value:X)H = $value"
```

Shift operators are also provided. Shift left “<<” and shift right “>>” allow moving bits at an arbitrary position:

```
for power in [0:16]
   value = 1 << power
   > @"2^$(power:2r): $(value:b17r)b = $(value:X4r)H = $value"
end
```

And, or, xor, shift left and shift right operators are also provided in the short assignment version; for brevity, and, or and xor assignments are respectively “&=”, “|=” and “^=”, while to avoid confusion with relational operators shift left and shift right assignments are indicated with “<<=” and “>>=”.

```
value = 0xFF00        // set an initial value
value &= 0xF00F      // Try an and...
> @"$(value:X)H"     // shall be F000H

value >>= 4        // drag a semibyte to the right
> @"$(value:X)H"     // shall be F00H
```

The functions

Functions are pieces of code that may be reused again and again by providing them with different values called parameters . More formally, functions are relational operators that relate a set of zero

or more values (called parameters) that can be taken from a finite or infinite set of possible values (called a dominion) with exactly one item that can be taken from a finite or infinite set of possible values (called a co-dominion).

Meet our first function:

```
function do_something( parameter )
   printl( "Hey, this is a function saying: ", parameter )
end

do_something( "Hello world" )
do_something( 15 )
/* again, ad libitum */
```

Functions are declared by the `function` keyword, followed by a symbol and two parenthesis which can contain a list of zero or more parameters. In this case, the function doesn't return any value; actually this is an illusion, because a function that does not return explicitly a value will be considered as returning the special value `nil`.

Functions can even be declared after being used:

```
do_something( "Hello world" )
do_something( 15 )

function do_something( parameter )
   printl( "Hey, this is a function saying: ", parameter )
end
```

All the code that is not in a function is considered to be "the main program"; function lines are kept separated from normal code, and so it is possible to intermix code and functions like this:

```
do_something( "Hello world" )

function do_something( parameter )
   printl( "Hey, this is a function saying: ", parameter )
end

do_something( 15 )
```

Anyhow, it is very important to have a visual reference to where the "real program" begins, if it ever does, so in real scripts you should really take care to separate functions from the other parts of the script, and show clearly where the function section or main section begins:

```
/*
   This is my script
*/

do_something( "Hello world" )
do_something( 15 )

/************************************
       Main section over,
       starting with functions.
************************************/

function do_something( parameter )
   printl( "Hey, this is a function saying: ", parameter )
end
```

Or if you prefer a bottom-top approach:

```
/*
   This is my script
*/

function do_something( parameter )
```

```
   printl( "Hey, this is a function saying: ", parameter )
end

/************************************
       Main program begins here.
************************************/

do_something( "Hello world" )
do_something( 15 )
```

As many other statements, functions executing just one statement may be abbreviated with the colon indicator (":").

Functions are not just useful to do something, but also to return some values:

```
function square( x )
   y = x * x
   return y
end
```

or more briefly:

```
function square( x ): return x * x
```

Return will immediately exit the function body and return the control to the calling code. For example:

```
function some_branch( x )
    if x > 10000
      return "This number is too big"
    end

   /* do something */
    return "Number processed"
 end
```

The first return prevents the rest of the function to be performed. It is also possible to return from inside loops.

A function may be called with any number of parameters. If less than the declared parameters are passed to the function, the missing ones will be filled with `nil`.

Recursiveness

Functions can call other functions, and surprisingly, they can also call themselves. The technique of a function calling itself is called recursion. For example, you may calculate the sum of the first N numbers with a loop, but this is more fun:

```
function sum_of_first( x )
    if x > 1
      return x + sum_of_first( x - 1 )
    end

    return x
end
```

Explaining how to use the recursion (and when to prefer it to a loop) is beyond the scope of this document but if you're interested here is a start point.

Local and global variable names

Whenever you declare a parameter or assign variable in a function for the first time, that name

becomes "local". This means that even if you declared a variable with the same name in the main program, the local version of the variable will be used instead. This prevents accidentally overwriting a variable that may be useful elsewhere; look at this example.

```
sqr = 1.41

function square( x )
   sqr = x * x
   return sqr
end

number = square( 8 ) * sqr
```

If the `sqr` name inside the function were not protected, the variable in the main program would have been overwritten.

Global variables can be accessed by functions, but normally they cannot be overridden. For example :

```
sqr = 1.41

function square( x )
   printl( "sqr was: ", sqr )
   sqr = x * x
   return sqr
end

number = square( 8 ) * sqr
```

will print `1.41` in the `square` function; however, when the `sqr` variable is rewritten in the very next line, this change is visible only to the function that caused the change.

Anyhow, sometimes it's useful to modify to an external variable from a function without having that variable being passed as a parameter. In this case, the function can "import" the global variable with the keyword `global` .

```
function square_in_z( x )
   global z
   z = x * x
end

z = 0
square_in_z( 8 )
printl( z )               // 64
```

Static local variables and initializers

Sometimes it's useful to have a function that remembers how its variables were configured when it was last called. Using global imported names could work, but is inelegant and dangerous as the names may be used to do something beyond the control of the function. Falcon provides a powerful construct that is called a "static initializer". The statements inside the static initializer are executed only once, the first time the function is ever called, and the variables that are assigned in it are then "recorded", and they stay the same up to the next call.

This example shows a loop that iteratively calls a function with a static initializer. Each time it’s called, the function returns a different value:

```
function say_something()
    static
       data = [ "have", "a", "nice", "day" ]
       current = 0
    end
```

```
    if current == len( data )
       return
    end

   element = data[current]
   current += 1
   return element
end

thing = say_something()
while thing != nil
   print( thing, " " )
   thing = say_something()
end
printl()
```

The first time say_something is called, the static statements are executed, and two static variables are initialized. The static block can contain any statement, and be of any length, just any local variables that are initialized in the static block will retain their value across calls.

We have also seen the nil special value in action; when the function returns nothing, signaling that it has nothing left to say, the `thing` variable is filled with the nil special value, and the loop is terminated.

Anonymous and nested functions

Normally, functions are top-level statements. This means that the "parent" of a function must be a Falcon module. Anyhow, on need however it is possible to create locally visible functions.

The keyword `innerfunc` can be used to define a new nested function that spawns from its definition to the relative `end` keyword. The function must be immediately assigned to a local or global variable:

```
var =  innerfunc ( [param1, param2, ..., paramN] )
   [static block]
   [statements]
end
```

The following is a working example:

```
square = innerfunc ( a )
    return a * a
end

printl( "Square of 10: ", square( 10 ) )
```

Anonymous functions can be nested, and generally manipulated as variables, as the following example demonstrates:

```
function test( a )
   if a == 0
      res = innerfunc ( b, c )
            // nested anonymous function!
            l = innerfunc ( b )
                return b * 2
            end
            result = b * l(c) + 1
            return result
         end
   else
      res = innerfunc ( b, c ); return b * c -1; end
   end

   return res
end
```

```
// instantiates the first anonymous function
func0 = test( 0 )

// instantiates the second anonymous
func1 = test( 1 )

printl( "First function result:", func0( 2, 2 ) )
printl( "Second function result:", func1( 2, 2 ) )
```

Anonymous functions are useful constructs to automatize processes. Instead of coding a workflow regulated by complex branches, it is possible to select, use and/or return an anonymous function, reducing the size of the final code and improving readability, maintainability and efficiency of the program.

As with any other callable item, anonymous functions can also be generated by a factory function and shared across different modules.

To exploit this feature to its maximum, it is important to learn to think in terms of providing the higher levels (the main script) not just with data to process, but also with code to perform tasks.

Function closure

Function closures, or nameless functions , act as inner functions that may cache the value of the local variables (and parameters) of the context in which they were declared. They are declared through the `function` keyword, and work exactly as "inner functions", except for the fact that they perform a closure on their context.

Take the following example:

```
function makeMultiplier( operand )
   multiplier = function( value )
      return value * operand
   end

   return multiplier
end

m2 = makeMultiplier( 2 ) // ... by 2
> m2( 100 )              // will be 200

m4 = makeMultiplier( 4 ) // ... by 4
> m4( 100 )              // will be 400
```

At the moment, closure can access and close only the local symbols in the direct parent. They won't close symbols from the global scope nor from any other surrounding function other than their parent's; however is it possible to repeat a global variable or an outer scoped variable to the immediate parent by simply naming it to have it reflected, as in the following sample:

```
function makeMultiplierGlobal()
   global globalFactor

   g = globalFactor    // repeat locally

   multiplier = function ( value )
      return value * g
   end

   return multiplier
end

globalFactor = 2
m2 = makeMultiplierGlobal()    // ... by 2
> m2( 100 )                    // will be 200
```

```
globalFactor = 4
m4 = makeMultiplierGlobal()    // ... by 4
> m4( 100 )                    // will be 400
```

Inner functions and nameless functions are actually expressions. The first example may be rewritten as:

```
function makeMultiplier( operand )
    return ( function ( value );
      return value * operand;
    end )
end
```

Notice the extra ";" after each line. In an open parenthesis context, EOL is not considered a statement terminator anymore; the termination of the statement must be explicitly indicated through the semicolon symbol. As such, the whole statement may be rewritten on a line:

```
function makeMultiplier( o ): return ( function (v); return v * o; end )
```

Codeblocks

Codeblocks are mainly a syntactic sugar for nameless functions, especially for functions just needing to perform an expression and return its value.

The formal declaration of lambda expression is the following:

```
block = { [p1, p2..., pn] => expression }
// or

block = { [p1, p2..., pn] =>
 statement
 ...
}
```

where `p1 ... pn` is an optional list of parameters.

Actual parameter values can be fed through the function call operator (open and close round parenthesis). For example, to print the sum of a number a rather complex way may be:

```
printl( {a, b => a + b}(2,2) )
```

Notice that codeblocks are functional closures, so:

```
function makeMultiplier( o ): return { v => v * o }
multBy2 = makeMultiplier( 2 )
> multBy2( 100 )                        // will be 200
```

Codeblocks containing more than one single expression are totally equivalent to nameless functions; as for nameless functions, it is necessary to return a value from within them to make it visible in the caller. For example:

```
posiSum = { a, b =>
   val = a + b
   if val < 0: return -val
   return val }

> posiSum( -5, 1 )   // 4
> posiSum( 5, 1 )    // 6
```

Callable arrays

When the first element of an array is a function, the array becomes "callable". Applying the function call operator to it, the function used as first element is called. Other elements in the array are passed to the function as parameters; other parameters may be passed to the function normally, through the call operator.

For example, the following code always prints "hello world".

```
printl( "Hello world" )
[printl]( "Hello world" )
[printl, "Hello"]( " world" )
[printl, "Hello", " ", "world"]()
```

It is then possible to create pre-cached parameters functions. For example, to prepend every call to `printl` with a prompt, the following code may be used:

```
p_print = [printl, "prompt> "]
p_print( "Hello world!" )
```

An interesting usage of arrays as functions with pre-cached parameters is that to store an item in the array by reference. Item references will be shown in a future chapter, but consider the following code:

```
i = 0
icall = .[printl $i ": "]
for i in [0:10]: icall( "Looping..." )
```

Notice that in this code we have used the dot-square array declarator to avoid using commas between tokens.

The "`$i`" code extracts a reference out from the "`i`" variable; the called function will be then presented with the current value of the desired variable, and not with the value that it had when the callable array was created.

Callable arrays are considered callable items in every aspect. For example, they can be used everywhere a function is needed as a parameter (I.e. the `arrayFilter()` function), or in place of methods for objects.

Accessing the calling context

At times, it is useful to know:

- Who is the caller of a function.
- In which function we're currently working.

The keyword `fself` assumes the value of the currently executed function. It is syntactically equivalent to use the name of the function in which the `fself` keyword is used. For example:

```
function sumFirst_1( n )
   return n <= 1 ? 1 : n + sumFirst_1( n-1 )
end

// this is equivalent
function sumFirst_2( n )
   return n <= 1 ? 1 : n + fself( n-1 )
end
```

The `fself` keyword comes handy when the name of the current function is not known, as for code

blocks and nameless functions.

```
> "Sum of first 30 numbers: ", { n => n <= 1 ? 1 : n + fself( n-1 ) } (30)
```

The `caller` method can be applied to functions to discover who called them.

```
function recurse( val )
    if val <= 0: return 1
    > recurse.caller(), ":", val      // or fself.caller()
    return recurse( val-1 ) + val
end

recurse( 5 )
```

We'll see that when the caller is a method, this comes quite useful as we can guess which object originated a call for us.

Non positional parameters

It is possible to address function parameters by names through a construct called *future binding*. In short, a parameter with a given name can be filled by adding a '|' (pipe) sign after a single-word symbol, and giving a value or an expression right after. See the following example:

```
function f( alpha, beta, gamma )
  > @"alpha: $alpha"
  > @"beta : $beta"
  > @"gamma: $gamma"
end

f( gamma| "a" + "b" + "c", beta|"b-value" )
```

This sets nil in alpha, "b-value" in beta and "abc" in gamma. The interesting point about "future bindings" is that they are actually expression; so the above `f` call is equivalent to:

```
future_beta = beta|"b-value"
future_gamma = lbind( "gamma", "a" + "b" + "c" )
f( future_gamma, future_beta )
```

The `lbind` function can create *late* and future bindings; it's less efficient than the pipe operator, but it's more flexible as it can create bindings with arbitrary names.

A future binding value gets expanded in any function call, including callable array calls. Calling a function with a future binding having a name not matching any parameter name raises an error, like in the following case:

```
   f( non_existing|"value" ) // raises an error!
```

It is possible to mix positional and non-positional parameters in the same call. In that case, the positional parameters are applied first, then all the non positional parameters (no matter where they appare in call order) are applied to the matching parameters. This may lead to effectively overwrite a value given in a positional parameter, as in the following example:

```
   f( beta| "new value for beta", "in alpha", "in beta" )
```

As the first positional parameter is stored in alpha, the second in beta, and then the beta parameter is overwritten by the named parameter, the result is the following:

```
alpha: in alpha
beta : new value for beta
gamma: Nil
```

Functional programming

According to Wikipedia's definition:

> *Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast with the imperative programming style that emphasizes changes in state.*

As Falcon provides a hybrid model, where the user can chose "how much functional" its code will be, we present some novel nomenclature to identify functional programming entities. Experts will recognize well known concepts from lambda calculus and former languages as LISP or Scheme being given new names, but as some concepts fit while other are slightly overlapping or not fully compliant with pre-existing theory, we decided to present a completely new nomenclature to avoid confusion. Otherwise, we should have borrowed names and indicated where they fit and where not. Moreover, the relatively novel concept of "optional" functional programming, where in the past it has been more or less the only paradigm of programming languages implementing it, requires a slightly different approach that justifies new names even for old things.

> *Before going into the new nomenclature, a bit of context information may be useful to those ones knowing functional programming. Falcon functional programming is an "impure" model (functional with "side effects") with strict evaluation. Actually, evaluation is "strict", that is, parameters are evaluated before functions, only if not explicitly required differently by the user through "constructs". Despite being impure, Falcon supports monadic programming. It is inherently possible to create pure functions with monads, and have various form of lazy evaluation, but this is currently not used by the VM to optimize calculus and is not "forced". Means to achieve this are just provided to the programmers for their usage.*

The theory

In this paragraph we explain formally how functional paradigm works in Falcon. In the next paragraph, we'll proceed with samples and introduce terminology little by little, so it is not necessary to read and understand this paragraph to proceed further. However, it is strongly advisable to gain confidence with the formalization of the paradigm, as understanding its working model may spare from some misunderstanding and headaches in using this features. In case this paragraph seems too abstract, it may be a good idea to start from the example and return to read it when the practice is mastered.

In short, the model used by Falcon functional programming is called Sigma to Alpha. The functional evaluation is the process to apply the Sigma-to-Alpha function to a list of zero or more items, each of which is called "topic". A topic may be either an Alpha, in which case Sigma-to-Alpha applied to that topic gives the same Alpha, or a Sigma, in which case it is Sigma-evaluated to obtain a result topic. The result may be an Alpha or again a Sigma, in which case Sigma-to-Alpha may be applied again until an Alpha result is reached. A list of topics is an Alpha if and only if each elements is an Alpha; this implies that topics containing a list of topics have Sigma-to-alpha applied to each one of their elements.

Sigmas are sequences composed of an application Kappa followed by zero or more topics. Sigma-evaluation of a Sigma is applying K to its argument (the list of topics), each of which being Sigma-

to-alpha processed before K -application.

Applying the Sigma-to-alpha function to a sequence of topics is called Sigma-evaluation, and applying a K to its topics is called K-reduction.

Let's discuss this topic in a bit less cryptic language. In functional context, we want to determine the final value of a sequence of data, which may contain functions or terminal data. "Determining" this value may be interesting both for the value itself or for the process that is performed to achieve the value, as this may involve calling functions. However, let's focus on determining the value. We have an item that must be evaluated in functional context (a topic), or sigma-evaluated . It may be itself an atom, or a list of atoms not needing simplification. Those two entities are indicated with the term "alpha", initial letter of "atom" in the Greek Alphabet.

In a sequence, the sigma-evaluation is applied to each element to determine if it can be reduced, while if the topic was not a list in the first instance, the value is already known and corresponds to the value itself.

If the first element of a sequence to be sigma-evaluated is a Falcon callable item (Kappa for Greek initial of "callable"), then the sequence must be simplified (hence the name "Sigma" from the Greek letter "S"). In that case, the other topics in the sequence are themselves Sigma-evaluated, and when they are all reduced, the list of topics is passed to the K callable symbol. The return value is substituted with the previous Sigma in the sequence where it was found.

The return value of a K-reduction needs not to be an alpha. It may be still a Sigma, in which case the topic may be evaluated again, if needed and wished.

A special class of K applications is called Eta (extra-Kappa). The K reduction of Etas modifies the way Sigma-evaluation is performed. Actually, when a Sigma-evaluation finds a Sigma whose K is an Eta, it doesn't sigma-evaluate the other topics in the list, and passes them unchanged to the Eta for K-reduction. Then, it's the Eta that, if needed, starts a subsequent Sigma-evaluation on its parameters, or on just some of its parameters. This is called Eta-reduction, and form the basis of the implementation of functional "constructs" in Falcon.

Actually the concept of Sigma currently overlaps 1:1 with the callable arrays, as the arrays are the only sequence known as Sigma at this stage of development, but we'll keep the term "sequence" through, as the model shall be extended to various kind of sequences.

Evaluating in functional context

Falcon, by default, is not in functional mode. Functional evaluation, or Sigma-evaluation, is invoked by using special functions that instruct the VM to begin to consider items in Sigma-evaluation mode. Those special functions are called "ETA" (from the Greek letter "eta"), and have a special significance for Falcon functional model. When the Sigma-evaluation meets an Eta function, the VM drops sigma mode and lets the Eta function handle the situation. Then, the Eta usually instructs the VM on further sigma-evaluation needed.

The most simple eta function is the `eval`, which sigma-evaluates its contents and returns the evaluation result.

```
> eval ( 1 )
```

Not surprisingly, this results in 1; in fact, 1 is an atomic value, whose sigma-evaluation value is exactly itself. Sigma-evaluation recognizes special values only to certain sequences, and specifically to callable arrays.

Sigma-evaluation on normal arrays results in sigma-evaluation of each element:

```
inspect( eval( [1,2,3] ) )
```

will result in [1,2,3]; however, consider the following case (we'll be using dot-square notation for arrays from now on):

```
function returnTwo()
   return 2
end

inspect( eval( .[ 1 .[returnTwo] 3 ] ) )
```

As the sigma-value of an array is the sigma-evaluation of each element, the `[returnTwo]` array gets evaluated. It's not a simple array; the evaluator finds out that its first (and only) element is a callable item, and to reduce it to a plain value the function gets called. The return value is substituted in place of the function that was called, so the final evaluation results in `[1,2,3]`.

A callable item as first element of an array is called Kappa, and calling the Kappa with its parameters is called K-reduction.

When the K is not an Eta function, that is, if it's a plain function that has no special meaning in functional evaluations, all its parameters get sigma-evaluated before it is called. For example :

```
function sum(a, b)
   return a+b
end

inspect( eval( .[
      .[sum .[sum 2 2] 4]
      .[sum 3 .[sum 3 3]]
      ]))
```

In this case, the inner sums are K-reduced and then they are passed to the outer sum for K-reduction. The result of this evaluation is an array [8, 9].

Another common Eta function is `iff` (functional if). This function Sigma-evaluates the first parameter. If the result is true, according to Falcon truth test, then the second parameter is Sigma-evaluated and the result of the evaluation is returned. If it's false, an optional third parameter will be Sigma-evaluated and returned.

For example :

```
function greater( a, b ): return a > b

printl( iff (
  .[ greater .[random] 0.5],
  "you were lucky",
  "you were unlucky" ))
```

> *Falcon evaluates as true non-zero numeric values, non empty strings, arrays and dictionaries and any object. Zero, nil empty strings, arrays and dictionaries are considered false.*

Notice two important aspects of the Falcon functional model. First, it is an impure model, meaning that K-reduction may have "side effects". In other words, they may alter the state of the program (as in creating items), and they can have input or outputs. Second, normal expression evaluation is also available in functional context; it is not strictly necessary that every parameter of an Eta-function gets evaluated through Sigma-evaluation; it's just an option at the programmer's disposal.

Given these two features, the above code can be rewritten as:

```
iff ( random() > 0.5,
        .[printl "you were lucky"],
        .[printl "you were unlucky"] )
```

Expression evaluation is performed before Sigma-evaluation, and is performed on every parameter of any function, regardless of their being Eta-functions or not; this means that it is possible to prepare simple values using imperative programming, and use them in a later functional context, as the above example has shown. However, be careful not to confuse imperative evaluation with K-reduction. Rewriting the above code as follows:

```
iff ( random() > 0.5,
        printl( "you were lucky" ),
        printl( "you were unlucky" ) )
```

both the `printl` calls would be performed, and then the Sigma-evaluation of one of their return values would end up being returned by `iff`. It is unlikely that this is the desired effect.

At times, the caller is interested in having a Sigma expression returned for later evaluation rather than evaluated on the spot. The lit Eta-function just passes down its parameters, interrupting the Sigma-evaluation. For example, take the following code:

```
picked =  iff ( random() > 0.5,
        .[ lit .[printl "you were lucky"]],
        .[  lit  .[printl "you were unlucky"]] )
...
picked()
```

This behavior is actually replicated by the Eta-function `choice`, which works like `iff`, but doesn't Sigma-evaluate the second and third parameter, returning one of them unevaluated. For example:

```
choice( random() > 0.5,
        .[printl "you were lucky"],
        .[printl "you were unlucky"] )()  // notice the () call operator.
```

Evaluation operator

Since version 0.8.12, Falcon also provides an *eval operator*, formed by a cap-star sequence, which performs functional evaluations on sequences. It's an unary operator returning the item as-is if it's not callable, calling it if it's a function or evaluating it if it's a sequence:

```
>  ^*  1   // 1

function  test()
  > "Test"
    return  2
end

^*  test   // calls test()
inspect( ^* .[ 1 .[test] 3 ] )   // evaluates 1,...,3 sequence
```

This operator can be used only to initiate functional sequences, but it is more efficient than calling the eval function. It is also meant to retrieve values that may be either values or functions that return a value in an efficient way:

```
// Instead of this...

// Should I get a value or call a property giving me the value?
if  obj.prop.type() == FunctionType
   value = obj.prop()
else
```

```
   value = obj.prop
end

// ...you can just do
value = ^* obj.prop
```

Multiple evaluation

The Eta-functions `all` and `any` take a sequence of items as parameters. The `all` Eta-function returns true if the results of all the items in the sequence are true, while `any` returns true if anyone of the items is true. Each item is Sigma-evaluated separately; so the net effect of all and any in functional context is that to perform Sigma-evaluations on a sequence of items respectively while and until an evaluation result is true.

Consider this example:

```
function falsifier(): return false
function verifier(): return true

any( .[
   .[falsifier .[printl "First call..."]]
   .[verifier .[printl "Second call..."]]
   .[falsifier .[printl "Third arg..."]]
])
```

As falsifier is not an Eta-function, its arguments are sigma-evaluated, with the effect to print the desired string; then the falsifier is called (with a value that is ignored), and being false it forces the evaluation of the next statement. The verifier returns true; once Sigma-evaluated its parameters, the function interrupts evaluation, and the third element is not K-reduced.

There are also two Eta-functions named `allp` and `anyp` , which work respectively as `all` and `any` , but they doesn't take a sequence as unique parameter; instead, they repeat Sigma-evaluation directly on each of their parameter respectively while and until an item evaluates to false. They are provided to avoid redundant array declarations, especially when all and any are themselves part of Sigmas:

```
iff ( reverseCalc,
   .[ any .[third second first]],
   .[ anyp first second third] ) // notice anyp vs. any
```

These four little functions can be also used to shorten long and `if` and/or sequences. For example , an `if` statement like the following:

```
if cond1 and cond2 and cond3 and cond4
   ...
```

may be even more readable if written as

```
if allp( cond1, cond2, cond3, cond4)
   ...
```

and even long sequences of logical operators may be changed into all/any sequences:

```
if (condition(1) and condition(2) and (c1 or condition(3))) or \
   (c2 and condition(4))
...
end

// same, but using anyp/allp
if  anyp ( .[ allp .[condition 1] .[condition 2] .[ anyp c1 .[condition 3]] ],
         .[ allp c2 .[condition 4]])
```

```
...
end
```

Cascading

The Eta-function `cascade` evaluates a sequence of K-applications one after another, where the result of the first evaluation is feed as parameter of the second one and so on. In other words, `cascade` executes a predefined list of functions one after another, passing the result of the former as the parameter of the latter. The first K-application in the list receives all the parameters given to `cascade`, and the return result of the last K-application is finally returned by `cascade`. As an example, it's possible to use the classical mathematical definition of absolute value as the square root of the square of a number.

```
function square( x ): return x * x
function sqrt( x ): return  x**0.5
> cascade( .[square sqrt], 5 )      // prints 5
> cascade( .[square sqrt], -5 )     // prints 5 (again)
```

Cascade becomes useful when used create new functions by simply concatenating existing ones. For example , to create a "classical abs" function that works as the above example, the following code can be used:

```
cascade_abs = [cascade, [{ x => x*x}, {x => x ** 0.5} ]]
> cascade_abs( 5 )      // prints 5
> cascade_abs( -5 )     // prints 5 (again)
```

The items in the sequence of K-applications need not to be just K, they can be complete Sigma (that is, i.e. callable arrays), so it's possible to pre-cache parameters that will be passed to functions in the sequence before the value coming from previous evaluation. For example , the following sequence calculates the percent ratio between two numbers.

```
function factorize( factor, x ): return x * factor
function proportion( whole, part ): return part / whole
percent = .[cascade .[proportion .[factorize 100]]]

// if the whole is 2, and the part 1, what % is it?
> percent( 2, 1 ), "%"                                  // 50%
```

The `factorize` K has been given 100 as the first parameter; the second will be the one received as result of the `proportion` function.

Being faithful to the impure character of Falcon functional programming model, `cascade` accepts functions having a pure side-effect, that is, accepting the value or values given as parameters, but refusing to produce a result. If a function returns an out-of-band item, then it is considered as if not called, and the same parameter (or parameters) that were given to it by `cascade` are given to the next one. In the following example, we inspect results as they are being formed:

```
function factorize( factor, x ): return x * factor
function proportion( whole, part ): return part / whole

function inspector( phase )
   >> phase, ": "
    for id in [1:paramCount()]
      >> parameter( id )
      formiddle: >>", "
      forlast : > "."
    end
      // returns an out-of-band item
    return  oob( nil )
end
```

```
percent = .[cascade .[
   .[inspector "Begin"]
   proportion
   .[inspector "After proportion"]
   .[factorize 100]
   .[inspector "After factorize"]  ]
]

> percent( 2, 1 ), "%"
```

The `oob()` function turns any item into an *out-of-band* item; the out-of-band characteristic of an item is a marker, a signal that the upstream function wants special (namely out-of-band) processing for the item being returned. Scripts can use the out of band feature too. We'll talk more specifically of out-of-band items in a later paragraph.

When a function in the sequence returns an out-of-band item, it instructs `cascade` not to use its return value, and to pass the same parameters it has received to the next function.

List evaluation

Some functional programming oriented functions (some of them being Eta-functions, other being normal functions) operate repeatedly on a list of elements. They are: the normal functions `map` , `filter` and `reduce` and the Eta-Functions `dolist` and `xmap` .

The `map` function transforms a list into another through a so called "mapping function"; it returns a sequence where each item is the return value of the mapping function applied to the original item. For example , to obtain the square of three numbers using map, it is possible to do:

```
s1, s2, s3 = map( { x => x * x }, [1, 2, 3] )
```

The variables `s1` , `s2` and `s3` will respectively contain 1, 4 and 9.

The `filter` function stores the original items passed in an array in the result sequence if a "filtering" function applied to them returns true. For example , to filter a list so that only even numbers are left in, the following code can be used:

```
function passPair( x )
    return x % 2 == 0
end

vals = filter( passPair, .[1 2 3 4 5 6] )
inspect( vals )
```

The `vals` array holds now the values 2, 4 and 6.

The functions `filter` and `map` can be combined to filter an array and change its value, but `map` may be also used to filter the mapped sequence: if the mapping function returns an out-of-band item, the value is skipped. The example can be rewritten as:

```
function mapPair( x )
    if x % 2 != 0: return oob(nil)
    return x
end

vals = map( mapPair, .[1 2 3 4 5 6] )
inspect( vals )
```

The `reduce` function applies a sequence of values to a reducing function one at a time; it also passes the previous return value of the reducing function to the next call as the first parameter. An

extra initialization value can be optionally given; if present, that initialization value will be used as the first parameter in the first loop, else in the first loop the reducing function will be called with the first two elements in the array.

The final return value of the `reduce` function is the last return value of the reducing function.

For example , the `reduce` function may be used to sum all the numbers in an array:

```
function sum( x, y ): return x + y
> reduce( sum,           // the function
  [1, 8, 4, 9 , 3, 2],   // the array to be reduced
  0                      // the initial sum value
)
```

Using `reduce` in a bit of a smart way, it is possible to do interesting things such as calculating the mean value of a sequence:

```
function partialMean( x, y )
   static: count = 0

   // use an oob value to control start...
   if isoob( x )
      count = 0
      return y
    end

   // ... and to control the end
    if isoob(y): return x / count

   // normally, count and sum
   count++
   return x + y
end

function mean( sequence )
   return reduce( partialMean, sequence + oob(nil), oob(nil) )
end

> mean( [ 1,2,3,4,5,6,7,8,9,10] )
```

The `xmap` Eta-Function works as `map` , feeding a list of items as parameters of a given function one at time and building an array of results, but it has a relevant difference: it is an Eta-Function, so it takes care to Sigma-reduce its parameters by itself. Each item of the list to be mapped is Sigma-reduced before being fed to the mapper, and if the mapper is not a simple function, but a sigma, it is Sigma-reduced too before being used.

In other words, it is possible to map function results, and provide a variable mapper (the result of the sigma function). If one of the items in the list, or the evaluator itself, is not to be evaluated, it can be passed to xmap through the lit Eta-Function.

Finally, the `dolist` Eta-Function works similarly to `xmap` , feeding a Sigma-reduced item from a sequence in as parameter of a given function, but it doesn't create a result map. The `dolist` function is useful when it's necessary to process the list or to generate some non-functional effect in the processing function, avoiding paying extra unneeded memory. For example , the following example creates a even number counter:

```
function countPairValsIn( sequence )
   // local function cp
   cp = function ( x )
   static : count = 0

   if isoob(x)
      val = count
      count = 0
      return val
```

```
   end

   if x%2 == 0: count++
      return  true
   end

   dolist ( cp, sequence )
   return cp( oob(nil ) )
end

> countPairValsIn( [1, 2, 3, 4] )  // will print 2
```

Late bindings

So far we have been using external item references to provide functional evaluations with symbols that may be changed. Although effective, this technique precludes the possibility of creating a single execution unit consisting only of functional code. Also, sharing code across coroutines (logically parallel program execution units) could be tricky.

Falcon proves an item type called "late binding" which is specifically designed to be bound to a value only during array-calls and functional evaluation. Late bindings can be generated at any time through the `&` operator and the `lbind` function.

For example :

```
counter = lbind( 'counter' )
> counter
```

Late bindings become interesting when merged with arrays. Arrays can be assigned local variables through the '.' operator; then, a late binding can be applied to retrieve that value and pass it to the function during the call or the evaluation.

For example :

```
calling = .[printl &value]
calling.value = "Hello world"
calling()                       // execute directly
eval( calling )                 // evaluate in functional context
```

So, the value local symbol will be bound only at call, and it may be changed between calls, or changed directly in between. For example :

```
function inc( x ): x+=1

calling = []                    // our execution canvas
calling.value = 10              // initializing
calling += .[ .[inc &value]]    // adding an increment call
calling += .[ .[inc &value]]    // and another one
eval( calling )
> "After increment: ", calling.value
```

The binding context is always the outermost sequence in a computation. Notice that the `calling` array in the above example includes two sub-arrays, both calling `inc` , but the `&value` late binding refers to `calling` and not to them.

As late bindings are items, they can be stored and applied separately.

For example :

```
calling = .[ printl ]
calling.even = "Even number: "
calling.odd = "Odd number: "
```

```
for i in [0:4]
   valsym = i % 2 == 0 ? &even : &odd
   (calling + valsym)( i )
end
```

In the above example, a callable array is first created, and two symbol values are stored into as `even` and `odd` . Then, one late binding, either `even` or `odd` , is set in `valsym` and a new array composed from the original and the applied late binding is created and called at each time. The result is:

```
Even number: 0
Odd number: 1
Even number: 2
Odd number: 3
```

Self-referencing local values.

Late bindings can have any value, including functions or callable sequences. For example :

```
sequence = .[ &multiplier 3 ]
sequence.multiplier = .[ {x => x * 3} ]
> eval(sequence)
```

Notice that when calling directly sequence in the above example is not possible as &multipler isn't bound to a callable item (the code block) until the evaluation context is started or the execution is performed.

Considering the fact that local values can store arbitrary functions, and that they can be used separately from the evaluation of the sequence, sequences can also be used to produce flexible object-oriented like functional operations. The special binding value `&self` resolves into the sequence leading the evaluation context, and can be used to reference the "calling sequence".

For example, it is possible to use part of the current functional sequence as a value:

```
load funcext           // for the "at" function

vector = .[.[printl .[at &self 1] ] 100]  // print [1] of this sequence
eval( vector )                            // 100
vector[1] = "New value"
eval(vector)                              // "New Value"
```

Of course, `&self1` wouldn't work as `&self` gets bound only during an evaluation, while using [] on it would try to access the current value of self.

Finally, the `self` keyword (notice the lack of '&') can be used to access the host sequence when calling methods created as functions bound in the sequence. The following example prints each item in vector.

```
vector = .[ 'a' 'b' 'c' 'd' ]
vector.display = {=>dolist( .[printl "Item... "], self ) }
vector.display()
```

In short: `&self` resolves in the currently being evaluated sequence (during evaluations), while `self` accesses the owning sequence in bound methods during calls.

Parametric evaluation

At times it becomes useful to modify the evaluated sequences through different executions. It is

generically possible to insert variable references in the sequences and modify their values, or to modify directly the sequences (by changing their contents, as they are arrays and can be modified), but the simplest way is to use late bindings, which allow to modify the atoms stored in a sequence as the evaluator reaches them.

One special kind of late bindings are the parametric bindings. They refer to parameters given to the evaluated sequence by the most internal `eval` call (or generically, a functional parametric evaluator), and are formed by the `&` symbol and the number of the parameter (1 based). So

```
eval( .[ printl ">" &1 "<" ], "The thing to be printed" )
```

Parametric evaluation comes very handy when building flexible functions on the fly:

```
looper = { limit, seq =>
   for n = 1 to limit
       eval( seq, n, limit )
   end }

looper( 5, .[ printl "loop " &1 " out of " &2 ] )
```

Functional loop

It is even possible to create loops using functional constructs. The `floop` (functional loop) Eta-Function takes a sequence given as parameter, and iterates forever calling each callable item in the list one after another. When the last callable item is processed, the first one is executed again. For example , the following loop prints forever "one, two, three".

```
floop( .[
   .[ print "first, "]
   .[ print "second, "]
   .[ printl "third."]
])
```

You can interrupt this endless loop by pressing CTRL+C. There are times when endless loops are needed, but usually, it is useful to provide a condition for the loop to terminate. If a function in the sequence returns an out-of-band 0, the loop is interrupted, and if it returns an out of band 1 the loop is restarted. In other words, out-of-band zero works as functional `break`, while out-of-band 1 is interpreted as functional `continue`.

The following example prints the numbers from 1 to 10.

```
i=0
.[floop .[
  .[inc $i]
  .[printl $i]
  .[iff .[ge $i 10] oob(0) ]
]]()

function inc(x): x = x + 1
function ge(x, y): return x >= y
```

First, an "i" variable is created with an initial value; then, the `floop` function is called iterating over a list of three Sigmas: the first increments its parameter, and gets called with a reference to "i", the second prints the variable and the third checks its value; if it's greater or equal than 10, an out-of-band 0 is returned, breaking the loop.

Notice that the "i" loop variable is always used by reference in the list. The list gets first created and then evaluated; when the variables inside the list may change during evaluation, it is necessary to use a reference to them, as the list gets created only once and then evaluated. Variables not passed

by reference are evaluated at list creation, or in other words, the evaluation access them by value using the value they had at list creation.

The same loop may be rewritten using code blocks instead of fully declared functions as follows:

```
i=0
.[floop .[
  .[{ x=> x=x+1 }  $i]
  .[printl $i]
  .[iff .[ {x => x>=10} $i] oob(0) ]
]]()
```

More functional loop

Another function performing functional loops is `times` . The `times` function is extremely flexible: it can perform ranged loops (as for/in) and it can either execute repeatedly functions or evaluate repeatedly sequences. It passes the current value of the loop variable as the first parameter of functions, or as the `&1` parametric evaluation parameter if receivs a sequence. As in `floop`, `oob(0)` and oob(1) have special meanings, causing the loop to be respectively broken or restarted.

For example:

```
// Times used with functions
times( 5, { x => printl( "As a function: ", x)} )

// Times used with sequences
.[times ,[2:11] .[                        // Notice the ',' to tell the range
  .[iff .[{ x=> x % 2 == 1}  &1] oob(1) ] // skipping impairs
  .[printl 'Counting even... ' &1]
]]()
```

The `times` function can be applied also to integers (will loop from 0 to the number excluded). It also provided as a method of both ranges and integers; so the above can be rewritten as:

```
// using the step variable
[2:11:2].times( .[printl 'Counting even... ' &1] )
```

Integers have also `downto` and `upto` methods, working similarly:

```
5.upto(10, { n => printl( "Loop up: ", n )} )
10.downto(5, .[ printl "Loop down: " &1 ] )
```

Out of banding in detail

It is often useful to give items a "special meaning", so that when they travel in functional sequence evaluations, or as we'll see, when they travel along in messages, or just when they are to be returned by functions in your scripts.

Every Falcon item can be associated with a out of band status flag which can be given, removed or queried through a set of three functions and four operators.

Operators are faster and possibly more compact and readable than functions, but functions are useful in functional sequences where the value of the item on which to perform the out of banding isn't ready or known from the beginning.

The operations on out-of-band items (oob) are the following:

- Assigning the out-of-band status: **`^+`** operator or **`oob`** function.
- Removing the oob status: **`^-`** operator or **`deoob`** function.

- Checking the oob status (evaluates to true if oob is given to the item): **^?** operator or **isoob** function.
- Reversing oob status (removing it if given, giving it if ungiven): **^!** operator.

The following example show a script-level usage of out of band items through operators.

```
function getNum()
   num = random(1,10)
   if num > 4: return ^+ num
   return num
end

for i = 1 to 10
   > "At round ", i, " the number is ", \
   ^? getNum() ? "less." : "greater."
end
```

In this example, the function getNum() also performs a check on the random number, and if it's greater than 4, it marks it as out of band via the ^+ operator. The calling program can check the outcome of this test via the ^? operator, and print a different string depending on what happened. This example is trivial, but in case the check is complex or if the remote code is the only part of the program that could easily perform the check (i.e. because the information needed to decide the status of an item are transient and destroyed soon after), then this technique comes extremely useful.

There are cases in which a function return, or a generic processing result, may legally be any Falcon item, including, for example, `nil`. In those cases, having an extra flag on the returned item if it is to be considered valid, of if the process evaluation was faulty or couldn't be completed correctly can be extremely handy and can obviate the need of setting validity flags elsewhere (in a by-ref parameter, in properties or in global variables).

While it is sometimes more appropriate to raise errors in those cases, in other cases the raise semantic may not be the correct solution to handle the caller the notion of an impossible evaluation. This is the case with functional sequences and message programming, where the caller may not be prepared to deal with exceptions coming from the processor code, but at times it also becomes useful in procedural and object oriented programming. In fact, a raised exception has the semantic value of signaling a problem, while an "empty result" or "result to be specially considered" may not necessarily be due to a fault in the processing algorithm or in the input data. It may just be one of the possible and legal outcomes of the processing.

Out of banding and procedural programming

Out of band construct comes also extremely handy in procedural programming. For example, functions with static data may reset or change their state through out-of-band paramters.

Suppose we want to create a generator, returning an item from an array. We can replenish the generator by sending it a new array, as in the following example:

```
function generator( data )
   static
      array = nil; pos = 0
   end

   if data
      array = data
      pos = 0
      return
   end

   if pos >= array.len(): return nil
```

```
   return array[pos++]
end

generator( [1,2,3] )
> generator()  // 1
> generator()  // 2
> generator()  // 3
> generator()  // nil...
```

Now, the generator may be a bit smarter, and allow to reset the internal counter if it receives an out of band integer number; let's change `if data` with...

```
   if isoob(data)
      pos = data
   elif data
     array = data
     pos = 0
   end
```

This new function can be invoked as follows:

```
generator( [1,2,3] )
> generator()  // 1
> generator()  // 2

generator( oob(0) )    // reset
> generator()  // 1
> generator()  // 2
```

The for/in loop understands generators; it respects formiddle and forlast blocks by caching in advance the results of generator calls, and terminates the loop when an `oob(0)` value is returned. For example, changing the `return nil` into `return oob(0)` in the above generator, it's possible to feed the function in a complete for/in loop:

```
function generator( data )
   static
      array = nil; pos = 0
   end

   if data
      array = data
      pos = 0
      return
   end

   if pos >= array.len(): return oob(0)
   return array[pos++]
end

generator( [1,2,3] )
for elem in generator
   forfirst: >> "Begin: "
   >> elem
   formiddle: >> ", "
   forlast: > "."
end
```

This will print "`Begin: 1, 2, 3.`"

Generators could be created also with closures, methods, callable arrays, or in general with any callable Falcon item returning an oob(0) element to terminate the loop.

Objects and classes

Falcon provides a powerful object model that includes multiple inheritance, caller object tracking (sender) one bit attributes setting and non-classed objects. Here, we'll discuss how Falcon objects and classes can be used.

> *Shortly we'll introduce Falcon objects before speaking about the more general cases of the classes. Objects are actually class instances whose class remains hidden in the virtual machine and is not accessible at script level.*

Falcon stand-alone objects

An object is an entity that possesses a series of properties. A property may be seen as a "variable" that belongs exclusively to a certain object. Some of these properties hold a function that is made to operate on the objects they belong to; these special functions are called methods. For example, an object modeling a cash box will have a property that records the current amount of money being held in it and a method to deposit and withdraw cash. So, supposing we have already our cashbox ready to be used, we may do natural operations on it like this:

```
cashbox.amount = 50   // initial amount

/* some code here */
object.deposit( 10 )

/* some code here */
object.withdraw( 20 )

/* some code here */
printl( "Currently, the cashbox holds ", object.amount, " Euros." )
```

The "." (dot) operator is also called the "object access operator", and is used to access a property or a method that is inside an object. Properties can written and read without any particular limitation; in some contexts, however, you'll prefer to have a method that knows how to handle the objects internally. For example, we can code a withdraw method that raises a warning if it finds the amount property has fallen below 0.

> *A sort of **protected** scope is provided when declaring properties starting with an underline _.*

The simple cashbox object may be declared as follows:

```
object cashbox
   amount = 0

    function deposit( qt )
      self.amount += qt
    end

    function withdraw( qt )
      self.amount -= qt
    end
end
```

Our object has a quantity of money (initially set to 0), and two methods: deposit and withdraw. In the `object` statement, every property is simply declared by assigning a constant (initial) value to it; the `object` declaration supports only the assignment from constant statement, other than the

method declarations. Methods are nothing more than the functions we have just seen; as the `object` statement body does not support function calls, the declaration keyword `function` is not necessary to state that a method is being declared; the name of the method followed by parenthesis, optionally including a parameter list, is enough to define it.

The cashbox example includes a new keyword: `self`. This keyword refers to the *object that is currently handled by the method*. It is necessary for every method to use the self object to access object properties: look at this example:

```
object cashbox
   amount = 0

    function  deposit ( qt, interest_rate )
      amount = qt * interest_rate
      self.amount += amount
    end

   /* other things */
end
```

As this example explains, methods may need local variables too, and they are simply defined by assigning some value to them. So, to distinguish between the "amount" local variable and the "amount" object property, the self object alias must be used.

Here follows a more formal definition of the object statement:

```
object object_name [ from class1, class2 ... classN]
   property_1 = expression
   property_2 = expression
   ...
   property_N = expression

   [init block]

    function method_1( [parameter_list] )
      [method_body]
    end
   ...
    function method_N( [parameter_list] )
      [method_body]
    end
end
```

The method declarations may be shortened with the colon sign (":") if they hold only one statement, exactly as the `function` declaration.

We'll see later on that the `class` statement has a much more flexible way to define object entities, but the `object` statement is meant to provide a fast but clean way to define simple objects that are unique in their instances.

Once an object is defined, it is not possible to add new properties or new methods. Anyhow, it is possible to change its methods and properties at will; look at this example:

```
function new_deposit( qt )
    if self.amount + qt > 5000
      printl( "Sorry, you are too rich to be a programmer" )
    else
      printl( "Ok, we authorize you to have that money" )
      self.amount += qt
    end
end
old_deposit = cashbox.deposit
cashbox.deposit = new_deposit
cashbox.deposit( 10000 )  // we are now forbidden to do that.

old_deposit( 10000 )      // but the old method still works
```

```
printl( cashbox.amount )  // will be initial amount + 10000
```

Function names can be seen as items, and assigned to a property (or to any variable in general); it does not matter if that property was originally a method or not. In fact, we may have even assigned a function to `cashbox.amount`, but in this context it would not make much sense. Also, it is possible to store a method from an object in a variable; the variable also records the object from which the method was from, and when used to call the old method it will feed the old object in `self`. Finally, as you can see, functions also may access the `self` object; it is possible that they are assigned to an object method, and in that case, the `self` item will assume the value of the original object.

If a function call is not performed via a method, that is, if it's a function that is just called as we have done before this paragraph, the `self` item will assume nil value. So, it is possible for a function to know if it's being called as a method or not:

```
/* Data definitions */

function maybe_method()
   if self = nil
      printl( "I am a function" )
   else
      printl( "I am a method" )
   end
end

object test
   property = 0
end

/* Main program */

maybe_method()                // prints "I am a function"
test.property = maybe_method
test.property()               // prints "I am a method"
maybe_method()                // prints "I am a function" again
```

We are going to see how to further configure the actions of possibly methods or object users in the next paragraph.

The "provides" and "in" operators for objects

As object structures may be configured independently from predefined formal definitions (also known as classes), it is necessary to provide a way to have some information about object internals at runtime, so that generic object users have a minimal ability to configure themselves depending on object structure. In advanced object oriented languages, it is generally possible to know which type of object is currently being handled; in Falcon, although this information is present, it may not be enough to manipulate objects. Some of them, in fact, are just "objects", to which any property may be attached.

The `provides` keyword is a relational operator that assumes the value of 1 (true) if a certain object (first operand) provides a certain property (second operand). Let's redefine the `new_deposit` function/method to act a little smarter. Provides never raises an error; it will just be false when the first operand is not an object:

```
function new_deposit( qt )
   if  not  self  provides amount
      printl( "This should have been a method of an object with an amount" )
      return
   end
```

```
   if  self.amount + qt > 5000
      printl( "Sorry, you are too rich to be a programmer" )
   else
      printl( "Ok, we authorize you to have that money" )
      self.amount += qt
   end
end
```

Now, if you try to call this function directly from the program, like this:

```
new_deposit( 1000 )
```

the function will refuse to access the self object, which in fact does not exist.

We have already seen the `in` operator in action: it's the relational operator that scans an item for some contents: substrings in strings, items in arrays and keys in dictionaries. The in operator is so flexible that it can also be applied to objects and properties.

While the `provides` operator is specifically meant for objects, and is able to understand that its second operand should be a property, `in` is more generic and doesn't have this ability. As a side effect, `in` is slower than `provides`, but it is more flexible. The `in` operator will check for a string to be the same as a property name in the second operand, if it is an object; the function we have just seen may be also be written as:

```
function new_deposit( qt )
   if not "amount" in  self
      printl( "This should have been a method of an object with an amount" )
      return
    end
   /* the rest as before */
end
```

One first thing to note is that `in` would also work if the second operand were a string, an array or a dictionary, in which "amount" may be found. So, except for the special case of `self` which may only be an object or nil, it is necessary to check for the second operand to be an object (or to be sure of that) before trying to use `in`. The other thing to be noted is that, as `in` does not takes the value of the first operand as a literal, the value to be checked may be also set in a variable:

```
function new_deposit( qt )
    if some_condition
      property = "amount"
    else
      property = "interest"
    end

    if  not property in  self
      ....
    end
...
end
```

The init block

As you have probably noticed, the `[init block]` that has been introduced in the formal `object` declaration has not yet been explained. Objects can have an initialization sequence; however the explanation about how to use the `init` block in objects must be postponed, as it is first necessary to introduce the classes.

Classes

Defining an standalone object can be useful when there is specifically the need of one exact single object doing a very particular task; they are useful when the modeled object is so different from the other ones that it have very little in common with them (or nothing at all). However, in real world programs, this is a very rare case. Most of the time, there will be many points in common among different objects; it's natural for the human mind to categorize the reality so that every item of its attention is framed into a class. A class is then a schema, a basic description of the characteristics (properties) and behavior (methods) of a possibly infinite set of objects that may be represented or nearly approximated with this schema. The objects having those characteristics are called "instances", and the process to create an object from a class is called "instantiation".

Falcon classes are a mix of data (which is mainly extracted by the compiler at compile time implicitly) and code that can be used to instantiate objects. Classes are defined this way:

```
class class_name[ ( param_list ) ] [ from inh1[, inh2, ..., inhN] ]
   [ static block ]
   [ properties declaration ]
   [init block]
   [method list]
end
```

The property list of a class contains the name of the class properties and their initial values. Any valid expression can be assigned to the properties. If there isn't a meaningful value for an expression at initialization time, just use the `nil` value. Actually, the virtual machine and the compiler will work jointly to minimize the required initialization time, so that properties being given nil are not really assigned a value in the virtual machine loops during object creation, sparing time.

```
class mailbox( max_msg, is_public )
   capacity = max_msg
   name = is_public ? "" : "none";
   messages = []
   a_property = nil
end
```

The instantiation process can be performed in two ways. One is that of calling the class as it were a function; the effect will be that of create an empty object that will be fed in the class constructor, like this:

```
my_box = mailbox( 10, false )
printl( "My box has ",
         my_box.capacity, "/",
        len(my_box.messages),
        " slots left.")
```

The other way is that to use the object semantic to expand a base class or initialize normally uninitialized members:

```
object my_box from mailbox( 10, false )
   name = "Giancarlo"
end
```

As classes are considered generically "callable items", it is possible to manipulate them more or less like if they were functions. For example :

```
if some_condition
   right_class = mailbox
else
   right_class = X_mailbox
end
```

```
/* some code here */

my_box = right_class( 10, false )
printl( "My box has ",
         my_box.capacity # len( my_box.messages ),
         " slots left.")
```

To define a method in a class, the function keyword must be used:

```
class mailbox( max_msg )

   capacity = max_msg * 10
   name = nil
   messages = []

   function slot_left()
      return self.capacity - len( self.messages )
   end
end
```

At times, the initialization of an object requires a little more than just assigning some expressions to properties. In the cases, there are two options. One is to write a method that will be called separately to complete the initialization of the object once it is already created. Although this is a clean way to do the job, this may be painful when creating singleton objects. The other method is to use the *explicit initializer*, also known as *explicit constructor*, the init block.

The explicit initializer works as a method, with the exception that it is called during object initialization, right after property assignments; it receives the parameters that are declared in the class statements, so it cannot be given any other parameter declaration. This is an example:

```
class mailbox( max_msg )

   capacity = max_msg * 10
   name = nil
   messages = []

    init
      printl( "Box now ready for ", self.capacity, " messages." )
    end

    function slot_left()
      return  self.max_msg - len( self.messages )
    end

end
```

Properties can be declared static. By declaring a static property, it becomes shared among all the instances of a class. As the property is shared among many objects, it should not be initialized with dynamic data. The property will be initialized exactly the first time an object of that class is created; any update to the property will take effect both into all existing objects of the same class and into the new instances that are created afterwards.

```
class with_static
    static  element = "Initial value"

    function getElement(): return self.element;
    function setElement( e ): self.element = e;
end

obj_a = with_static()
obj_b = with_static()
printl( "The initial value of the property was: ", obj_a.getElement() );
obj_a.setElement( "value from A" )
obj_c = with_static()
printl( "The value in B is: ", obj_b.getElement(), " and in C: "
                         , obj_c.getElement()  );
```

The `init` block can have a `static` block that works very like function static blocks. However, the `init static` block is only called the first time an object of a certain class is instantiated. This allows preparing setup of an environment that will be then used by objects of the same class.

```
class with_static_init
   static numerator = nil
   my_number = nil

   init
      static
          self.numerator = 1
          printl( "Class initialized" )
      end
      self.my_number = self.numerator++
   end
end

obj_a = with_static_init()
obj_b = with_static_init()
obj_c = with_static_init()
printl( "Object number sequence: ", obj_a.my_number, " ",
                   obj_b.my_number, " ", obj_c.my_number )
```

The output will be:

```
Class initialized
Object number sequence: 1 2 3
```

Methods with static blocks

Static blocks can be declared in methods as they can be declared in normal functions; however, their behavior can be a bit counterintuitive. A method with a static block will enter and perform it only the first time the method is called for every instance of the parent class. Similarly, variables declared in the static block of a method are actually class static, and they are modified by all the instances of a certain class.

If there is the need to execute a part of a method only the first time that method is executed for a certain object, use a property, or even better, an attribute (see page74) and check its value with a normal branch.

Classwide methods

It's also possible to access a method from inside a class. Instance-less methods, or classwide methods, can be called directly from the class names, but, as they do not refer to any instance, they cannot access the `self` object.

Declaring and accessing methods directly from classes can be a simple way to define a **namespace** where to access normal functions. For example :

```
class FunnyFunctions
    function a()
       > "This is funny function a"
    end

    function b()
       > "This is funny function b"
    end
end

FunnyFunctions.a()
FunnyFunctions.b()
```

Classwide functions can be seamlessly merged with normal instance-sensible functions; the only requirement for a function to be callable not just from an instance, but also from the class, is that id doesn't access the `self` item.

Property accessors

Since 0.9.4.4

Accessors are hidden methods that intercept access on properties. Falcon accessors work on "virtual" properties. In short, you don't have to declare any property for the accessors to work on that; just declare the accessors for read, write or both the operations, and a "phantom" property will be created for you.

Write accessors are declared by creating a function named `__set_`*`propname`*, while read accessors are declared with `__get_`*`propname`*. It's two underline characters "_" followed by "set" or "get", another underline and the property name you want to mask.

For example, this creates an accessor for the `mean` property, which returns the mean of a series.

```
class Series( values )
   values = values

   function __get_mean()
      sum = 0
      for i in self.values: sum += i
      return sum / self.values.len()
   end
end

s = Series( [14,53,18,8] )
> "The mean is: ", s.mean        // 23.25
```

As there isn't any `__set_mean` function declared in the class, `mean` is considered a read-only property; trying to set it to a different value will cause an AccessError to be raised.

The `__set_` accessor receives as a single parameter the value that should be set in the property. The following code ensures that the property `value` is a number in the 0..100 range.

```
class Guarded( value )
   _value = 0

   // We can't declare a "value" property, but...
   init
      // we can initialize it via accessors here:
      self.value = value
   end

   function __set_value( v )
      if v.typeId() != NumericType: raise "Assigned not numeric value to 'value'"
      if v < 0 or v > 100: raise "Assigned value out of range to 'value'"
      > "Setting value ", v
      self._value = v
   end

   function __get_value(): return self._value
end

g = Guarded( 10 )
> "Initial value: ", g.value

g.value = 30
> "Value is now: ", g.value
```

It is also possible to declare just the `__set_` accessor for a given property. In that case, the

property becomes "write only" and can't be read back. This can be useful to create "plug points" that alter the host object depending on the data they receive, like in the following example:

```
class WOnly()
   data = nil

   init
      self.randomSeed = (seconds() * 1000) % 1000
   end

   function __set_randomSeed( seed )
      randomSeed( seed )
      self.data = random()
   end
end

> "New random number: ", WOnly().data
```

> *Notice that it's not possible to create a real property with the same name of a property guarded through accessors. Doing so will cause a Syntax Error for duplicate property name in class declaration to be raised.*

Multiple inheritance

One class (or one object) can be derived from multiple classes. The resulting class (or object) will have all the properties and methods of the subclasses. For example:

```
class parent1( p )
    prop1 = p
    init
       > "Initializing parent 1 with - ", p
    end

    function method1(): > "Method 1!"
end

class parent2( p )
    prop2 = p
    init
      > "Initializing parent 2 with - ", p
    end

    function method2(): > "Method 2!"
end

class child(p1, p2) from parent1( p1 ), parent2( p2 )
    init
       > "Initializing child with ", p1, " and ", p2
    end
end

instance = child( "First", "Second" )
```

As the example shows, the initialization of the child class is performed after the initialization of its parents, which is performed following the order in which the inheritance are declared.

The instance will have all its functions and methods, and all the methods declared in the base and child classes.

Base method overriding

It is often desirable that subclasses change the behavior of a base class by re-defining some

methods. Creating a new method (or property) in place of a method (or property) with the same name and parameters declared in a subclass is called *overriding*.

Once overridden, all the references to that property or method will receive the new member provided by the subclass. Consider this example:

```
class base
   function mth(): > "Base method"
   function callMth(): self.mth()
end

class derived from base
   function mth(): > "Derived method"
end
```

When an instance of the base class is created, it's mth method member is the function that prints "Base method". When an instance of the derived method is created, its mth method prints "Derived method". The derived instance inherits the callMth method. As now the mth member is the one provided by the derived class, the call self.mth() in its body will actually call the method in the derived class.

Continuing the above example:

```
base_inst = base()
base_inst.mth()           // prints "Base method"
base_inst.callMth()       // again, prints "Base method"

derived_inst = derived()
derived_inst.mth()        // prints "Derived method"
derived_inst.callMth()    // again, prints "Derived method"
```

It is however possible to access the base method (or property) of a derived object. For example, suppose that a class doesn't want to change a behavior of the base class, but just to extend it. Then it has to call the base class method before doing or after having done special processing. To access the method in the base class, the derived one must prepend the name of the base class to the name of the method (otherwise, it would call itself). In our case, the base class name is just base, so if we want to provide some callBase method in our derived class, it must be rewritten as follows:

```
class derived from base
   function mth(): > "Derived method"

   function callBase()
      > "pre-processing"
      self.base.mth()
      > "post-processing"
   end
end

derived_inst = derived()
derived_inst.callBase()
```

This will result in the base class method being called instead of the derived one. The base class methods and properties may also be accessed from the main code directly; it is possible to append the base class name after the instance name to reach the base class.

In the case of multiple inheritance, overriding may also happen among subclasses; if two or more subclasses have a method with the same name, the classes being instantiated last are the ones overloading former methods. In the following example, the mth method is offered by both subclasses:

```
class first
   function mth(): > "First method"
end
```

```
class second
   function mth(): > "second method"
end

class derived from first, second
end

instance = derived()
instance.mth()          // "second method"
instance.second.mth()   // again "second method"
instance.first.mth()    // "first method"
```

Private members

In object and class declarations, member names starting with the _ underline symbol can be accessed only through the self object. This means that only the object, or the class that declared them and its direct descendants, are able to access them. In case some of those properties or methods are needed outside the instance, they must be returned (or modified) using an accessor, that is a method returning or changing the property, as in the following example:

```
object privateer
   _private = 0

    function getPrivate(): return  self._private
    function setPrivate(value): self._private = value
end
```

However, private members should be used mainly to store data that should not visible by other parts of the program; for example they may be used to store an internal state or counter.

Subclasses can access private members declared by parent classes, but they can access them only through the `self` object. It is not possible to access a private member through a base class; once overloaded, the instances can access only the topmost overloading of the private member.

Operator overloading

Falcon objects (and class instances) provide several callback hooks that are called when the virtual machine applies operators on them. They all work similarly, by providing a method in the class or object definition, and they all are relative to *things done to the target instance*. In short, the virtual machine looks at the type of the first operand in an expression; if the operators are left-associative (i.e. +, *, - etc), the first operand is the leftmost. If that operand is an object or instance, and if it provides an appropriate overloaded method, that method is called and the other operands are passed as parameters.

For example, that's how "+" operator can be overridden in a class:

```
class OverPlus( initValue )
   numval = initValue

   function add__( operand )
      return OverPlus( self.numval + operand )
   end
end

op = OverPlus( 10 )
nop = op + 10
> nop.numval         //: 20
```

> *Operator overloads are suffixed with "__" (two underlines) to indicate that they are special, and to provide some "namespace protection" as names like "add", "sub" and so on may be very common and used by developers for other reasons. However, the overloaded nouns are not prefixed with underlines because this would make them private, and there is no inherent reason to make them so. You may want to call the overloaded methods directly, and if you can invoke them anywhere from the code through language level symbols, there is no meaning in removing the ability to call them or refer to them directly away from you.*

Of course, a class may want to check the type of the operand, and eventually work gracefully with compatible types. For example:

```
class OverPlus( initValue )
   numval = initValue

   function add__( operand )
      if operand provides numval
         return OverPlus( self.numval + operand.numval )
      elif operand.typeId() == IntegerType or operand.typeId() == NumericType
         return OverPlus( self.numval + operand )
      elif
         raise "Invalid type!"
      end
   end
end

op = OverPlus( 10 ) + OverPlus( 10 )
> op.numval          //: 20
```

Operator overloads are not bound to be read-only. If consistent, it is possible also to modify the values in self, as in:

```
...
   function add___( operand )
      self.numval += operand
      return self
   end
...
```

Also, the return value is totally arbitrary; you may want to return a different value. This allows you to create so called *manipulators*, having special significance in expressions as in the following example:

```
object saver
   function add__( operand )
      > "Saving value... ", operand
      // save it to a file
      return operand
   end
end

persistent = saver + 100
> "data is ", persistent    // 100
```

> *This model has two limitations: first, it's not possible for the second operand to take control of the operator overloading. It's the first operand that is in control of overloading. Second, all the operator overloading is performed in atomic mode. This means that the virtual machine is not interruptible while inside those callback methods, and that functions that may interrupt or suspend the workflow of the virtual machine are forbidden. In short, calling `sleep()` from inside `add__` would raise an error, returning the VM to non-atomic mode and dumping all that's done inside*

callbacks.

Operators overloading is divided into the follow:

- mathematical operators overrides.
- comparison overrides.
- accessor ovverides.
- call overrides (functors).

Mathematical operator overloading

They are unary or binary operators overloading, working quite alike. Binary operators overloads gets a single parameter (the second operand) and are usually, but not necessarily, expected to return a value of the same type of self, or of the second operand:

- **add__** Overloads "+" operator.
- **sub__** Overloads "-" operator.
- **mul__** Overloads "*" operator.
- **div__** Overloads "/" operator.
- **mod__** Overloads "%" operator.
- **pow__** Overloads "**" operator.

NOTE: Since 0.9.6, "" is before the operators. So, we have "add", "__sub", etc.

Binary operator overloading works also when used as *self assignment*. For example, += operator calls the `add__` override and then stores the return value in the same variable that was used as self. For example:

```
class OperOver( num )
   val = num
   function add__(v): return self.val + v
end

o = OperOver( 10 )
o += 5
inspect(o)        // a numeric 15 value
```

Unary operators overloads receive no parameters; they are the following:

- **neg__** Overloads the unary prefix negation operator ("-" in front of a symbol).
- **inc__** Overloads the prefix "++" increment operator.
- **dec__** Overloads the prefix "--" decrement operator.
- **incpost__** Overloads the postfix "++" increment operator.
- **decpost__** Overloads the postfix "--" decrement operator.

Comparison overloading

Comparison operators, namely <, >, <=, >=, == and != all refer to the same overloaded method: `compare`. Notice the absence of the "__" suffix. This is both because of historical reasons and because `compare` doesn't exactly overload operators, but serve a more complex purpose with a different semantic.

The `compare` method is bound to return a number less than zero, zero or greater than zero if the self item is respectively less than, equal to or greater than the comparand item, passed as parameter. Contrarily to mathematical operators, the `compare` method should not raise an error in case the

items are not comparable: Falcon VM prefers to have a strategy to sort all the items, even when sorting has no physical reason. When the compare function hasn't any mean to determine a sorting order, it should return nil; this informs the virtual machine that the override gave up, and that the default ordering algorithm should be applied: items of different kinds are ordered based on the value of their typeId() methods, and items of the same kind are checked based on the place they occupy in memory.

```
class CmpOver( val )
   number = val
   function compare( oper )
      if oper provides number
         return self.number - oper.number
      elif oper.typeId() == NumericType or oper.typeId() == IntegerType
         return self.number - oper
      end

      // else let the VM do our work
      return nil
   end
end

ten = CmpOver( 10 )
> "Is ten > 5? ", ten > 5
> "Is ten != 3? ", ten != 3
> "Is ten <= 10? ", ten <= 10
> "Is ten > an array? ", ten > [1,2,3]
```

The `compare` method is not used just to resolve the comparison operators; it's used also when a default ordering method is needed, for example, in dictionary key insertions and searches, or in `arraySort()` call.

Important: as the compare method is invoked only on the first item of pair ordering checks, all the members of a collection to be sorted. For example:

```
function cmpFunc( o )
   if o provides number: return self.number - o.number
   return nil
end

class CmpOne( val )
   number = val
   compare = cmpFunc
end

class CmpTwo( val, name )
   number = val
   name = name
   compare = cmpFunc
end

dict = [ CmpOne( 10 ) => "ten", CmpOne( 5 )=>"five", CmpTwo( 7, "seven" ) => "seven" ]

for k,v in dict
   > @"$(k.number) => $v"
end
```

In this example, the items stored in the dictionary as (ordered) keys are different, but they all agree on the way they must be ordered (by sharing the same ordering function).

Subscript overloading

Access through the array subscript accessor () can be overloaded through the following methods:

- **getIndex__** Overloads in read mode. Will receive 1 parameter (index), and should return an item.

- **setIndex__** Overloads in write mode. Will receive 2 parameters (index, value), and it is supposed to return the stored item.

Be careful about the fact that the `index` parameter is not necessarily just a number; it may be any Falcon item, as the overrider may wish not just to implement an array semantic.

The following example implements a self-growing array, that enlarge itself when accessed out-of-bound.

```
class GrowArray( initSize )
   content = nil

   init
      if initSize
         self.content = arrayBuffer( initSize )
      else
         self.content = []
      end
   end

   function len(): return self.content.len()

   function getIndex__( pos )
      pos = self.absolutize( pos )
      return self.content[pos]
   end

   function setIndex__( pos, value )
      pos = self.absolutize( pos )
      return ( self.content[pos] = value )
   end

   function absolutize( pos )
      // for simplicity, consider only integer and not ranges.
      if pos < 0
         pos = abs( self.content.len() + pos )
      end

      if pos >= self.content.len()
         self.content.resize( pos+1 )
      end

      return pos
   end
end

arr = GrowArray()
arr[1] = "one"
arr[2] = "two"
arr[3] = "three"
inspect( arr.content )
```

Call overrides and functors

In Falcon, a function call is actually considered an expression with two operands: the called item and the parameter list. Objects can provide a `call__` method to override the call operator `(...)`. Objects implementing a `call__` method are called *functors*. Functors are *function objects*, that is, objects that have their own state and internal methods, but that can be called as regular functions.

In the following example, we use a functor to iterate through a vector, returning an oob(0) (functional break) when the scan is complete.

```
class gimmeNext( array )
   array = array
   pos = 0

   function call__()
      if self.pos >= self.array.len(): return oob(0)
```

```
      return self.array[ self.pos++ ]
   end
end

gn = gimmeNext( ["one", "two", "three"] )

while not isoob( data = gn() )
   > data
end
```

The interesting part, the `gn()` call, is actually resolved into the invocation of the `call__` method in the functor.

Parameters passed to the call functor are translated into parameters for the `call__` method:

```
object callTest
   function call__( a, b ,c )
      > "A: ", a
      > "B: ", b
      > "C: ", c
   end
end

callTest( 1, 2, 3 )
```

And of course, it's also possible to access the parameters through the variable parameter convention:

```
class promptPrinter( prompt )
   prompt = prompt

   function call__( /*var params */ )
      for i in [0:paramCount()]
         > self.prompt, parameter(i)
      end
   end
end

pp = promptPrinter( "::: " )
pp( "Hello", "world", "there!" )
```

Automatic string conversion

It is often useful to turn an object into a string representation; this is done also by the virtual machine in several occasions, as when adding the object to a string, or when printing through the ">" fast-print operator. To provide a suitable string representation, an instance may provide a `toString` method, returning a string.

For example:

```
object Test
   function toString()
      return "I am the test"
   end
end

// automatic transformation to string
value = "Represent me " + Test
> value

// or also directly
> "Directly: ", Test
```

Notice that

```
> Test + "..."
```

wouldn't work as expected: in that case, the `add__` override is used instead.

Object initialization sequence

Objects, or better, items automatically created with the `object` keyword, are actually instantiated from a class that is not accessible to the script. This class is synthetically created and stored in the module where the object resides; as the module is linked in the Virtual Machine, an instance is created as if the script called the constructor. So this code:

```
class my_class
   property = some_function()
      ...
end

my_object = my_class()
```

and this code:

```
object my_object
   property = some_function()
      ...
end
```

are nearly equivalent. Nearly... but not quite, because the constructor of the stand alone objects, and the complex initialization involving their property, is completely managed by the Virtual Machine before the first instruction of the main script is executed.

In other words, even if a module does not provide a main code, and even if it's not the main script, the VM may execute some code from it to initialize the objects it declares.

This provides a powerful and easy way to configure Falcon modules as they are loaded in the virtual machine. However, this may actually bring two orders of problems.

The first problem is that the VM must be correctly configured for debugging, limiting script execution time, controlling script memory consumption and so on before the script you actually want to launch is launched; at link time, VM may execute some code, or even a lot of code, so the VM must be set up correctly before the link sequence is initiated. But this is a problem that involves the embedders, and it's a bit outside the scope of this guide.

The second order of problem is that the order of object initialization is undefined. Of course, objects declared in a module A that is loaded by a module B are all initialized before the first object from B gets the chance to be initialized, but the order by which the objects in A are initialized is undefined.

Usually, this is not relevant for the user, unless some of the objects in the code refer to some other object in the module. Consider the following example:

```
object first
   list = [ " one " , " two " , " three " ]
end

object second
   sum = ""

   init
      for elem in first.list
          self.sum += elem
      end
   end
end
```

This code may work or not. By the time `second` is initialized, the list in `first` may have already

been initialized or not. It's a 50% guess.

Luckily, the VM does some work for us, in the forecast that an item may reference some other item in the initialization step.

First of all, before the first initialization is performed, all the items of the module are correctly created, and their methods are readied. In this way, accessing properties of an arbitrary object is possible, and its content will either be a valid method, `nil`, or the fully readied value, in case the object has already been initialized. Filling all the non-method properties with `nil` allows eventual callers, or the object itself, to understand if the initialization routine has been called or not.

The second thing the VM does for us is avoid re-initialization. So, if some property of some object is initialized externally before the VM has the chance to call the automatic initializer, the already provided value will be saved.

Given these two services, what the implementer must do to solve this situation is called *initialization on first use* idiom (this is actually C++ terminology, but also works great here).

The `init` method of `second`, referencing `first`, should be kind on the first object by calling some method of it that can set up the object in place of the automatic initializer. Even better, if possible access to the required property should be performed only via a method that is meant only for this reason. Those special methods are called *accessors*.

Here we see an accessor using the initialization on first use idiom in action.

```
 object first
   list = nil

   init
     // forces initialization if this has not still happened
      self.getList()
   end

   // accessor...
   function getList()
     // ... using init on first use idiom.
      if self.list = nil : self.list = [ " one " , " two " , " three " ]
      return self.list
   end
end

object second
   sum = ""

    init
      // Reading first.list via the accessor
      for elem in first.getList()
          self.sum += elem
      end
    end
end

printl( "second.sum: ", second.sum )
```

The elegant part of this idiom is that the necessity to use it is present only during the initialization step, that is, in the init block of objects, or in their property declaration clauses. Once the link step is performed, (provided the init block and/or the property declarations do their jobs correctly, as in the example for object first), the properties are correctly set up, and there isn't the need to go on using the accessors to read the properties from an object.

The correct initialization of objects can be performed by following these simple rules:

As said, the main module and objects in other modules don't need to access other object properties via accessors with init on first use idiom; however, be careful, as init routines and property

initialization clauses may call functions that in turn may reference uninitialized objects.

So, if possible, avoid creating objects that need other objects being declared in the same module during initialization phase. If you have to, if possible, isolate them in a separate module so that you know that those object are correctly initialized, or be prepared to use the accessor instead of the property within all the functions in the same module where the target object is located.

Classes in functional sequences

When evaluated in functional contexts, classes generate instances as if they were directly called. For example, you may fill an array of instances through a functional loop like the following:

```
class ABC( ival )
   id = ival
end

arr = []
1.upto(10, .[ arr.add .[ABC &1]] )
inspect( arr )
```

Stateful classes

Since 0.9.6

Some kind of problems are better solved through "agents" or "machines" which are invoked to respond to certain events or perform certain tasks, and provide a different behavior under different situations.

A scared bird

For example, let's take the example of a bird that needs to eat and is scared from people. The bird can be in one of three states: famished, quiet and scared.

Initially, the bird is "quiet".

```
class Bird
   famine = 0

   init
      self.setState( self.quiet )
   end

//...
```

The bird can chose, each time, to be idle, eat or flee.

```
   function eat()
       > "The bird eats."
       self.famine = 0
   end

   function idle()
       > "The bird is idle"
       ++ self.famine
   end

   function flee()
       > "Flap flap flap..."
       ++ self.famine
   end
```

We present the current situation to the bird via an "onEvent" callback, in which we'll tell the bird if there's food and/or people around.

When the bird is quiet, it will ignore food, and get scared if a person approaches. After three turns, it will get famished again.

```
     state quiet
      function onEvent( food_nearby, people_nearby )
         if people_nearby
            > "The bird is getting nervous."
            self.setState( self.scared )
         else
            self.idle()
            if self.famine > 3: self.setState( self.famished )
         end
      end
   end
```

When it's scared, it will flee when people is around; otherwise it will return to be quiet or famished depending on the famine level.

```
   state scared
      function onEvent( food_nearby, people_nearby )
         if people_nearby
            self.flee()
         else
            self.setState( self.famine > 3 ? self.famished : self.quiet )
         end
      end
   end
```

when it's famished it will eat and get quiet, if there is no people around. Otherwise it will get scared.

```
   state famished
      function onEvent( food_nearby, people_nearby )
         if people_nearby
            > "The bird is getting nervous."
            self.setState( self.scared )
         elif food_nearby
            self.eat()
            self.setState( self.quiet )
         else
            self.idle()
         end
      end
   end

// end class:
end
```

We can now test this code with a simple sequence, like the following:

```
quail = Bird()
for i in [0:12]
  food = random( true, false )
  people = random( true, false )
  > @ "Step $i: Food=$food, People=$people"
  quail.onEvent( food, people )
  > "The quail is in state ", quail.getState()
end
```

States definition

As we intuitively seen in the previous paragraph, states are subsets of alternative methods that can be active at any time on an object. Fromally...

```
class ...
  [class declaration]

   state <state name>
      [state declaration]
   end

    ...

   state <state name>
      [state declaration]
   end
end
```

Each *state declaration* consists of zero or more function definition. It is not necessary that all the states declare the same functions. If a state doesn't declare a method that is declared in another state, when entering it the undeclared method is left unchanged. So if state **A** declares method *one* and *two*, while state **B** declares only method *two*, when moving from state **A** to **B**, method *one* is left untouched.

Method names declared in states can be present also in the class definition. When a state is not yet applied, they are reachable and behave normally. Once a state redefining them is applied, the methods declared in the generic class part are *shaded* (not anymore reachable in the host object), but they are still available if accessed statically or through the base class name.

States are actually represented as string properties; so an expression like `className.state_name` where `state_name` is the name of a state, resolves exactly in a string "state_name".

The BOM methods `setState` and `getState` are normal methods. This allows to create synthetically state names and apply them at runtime.

Transition functions

Two special functions, called `__leave` and `__enter`, are called back respectively before a state is applied (with the previous state still active), and after entering the new state. State transition is performed as indicated in this pseudocode:

```
function setState( new_state )
   value = nil

   if self.__leave is callable
      value = self.__leave( new_state )
   end

   old_state = current state
   apply new_state

   if self.__enter is callable
      value = self.__enter( old_state, value )
   end

   return value
end
```

Both methods are optional and both can return a value (which is then returned by the setState method). `__leave` is called in the previous state, and receives the target state name as the parameter, while `__enter` is called after the new state is applied, and careceives the states that was previously active.

If both the methods are provided, `__enter` receives also the value returned by __leave as second

parameter; otherwise, if only __leave is provided, its return value is returned by setState to the caller.

In our bird example, we wanted to notify the user about the fact that the bird is getting nervous when it passes into the scared state. Adding this method to the scared state:

```
...
   state scared
      function __enter( origin, lr )
         > "The bird is getting nervous"
      end
      ...
   end
...
```

We can now be sure that this code is executed each time the state is entered.

> *Notice that controlling the return value of the setState() function it is possible to implement Mealy automata models (a mathematic definition of a finite state machine where output values obtained from input sequences are bound to state transitions).*

State inheritance

State definitions are are inherited subclasses. Subclasses can override parent classes states as a whole; an overridden state will allow the subclass to define a new set of functions, or none at all, in the given state.

In the following example the child inherits the **A** state, overrides the **B** state changing its functions, empties the **C** state and declares a new **D** state:

```
class Base
   state A
      function callme(): > "A from base"
   end

   state B
      function callme(): > "B from base"
   end

   state C
      function callme(): > "C from base"
   end
end

class Child from Base

   state B
      function callme(): > "B from derived"
   end

   state C
   end

   state D
      function callme(): > "D from derived"
   end
end

c = Child()
c.setState( "A" )
c.callme()

c.setState( "B" )
c.callme()

c.setState( "C" )   // still the same as B
```

```
c.callme()          // as C state is deleted

c.setState( "D" )
c.callme()
```

Multiple inheritance is applied to states with the same rules as it is applied to method and properties: the rightmost child values override the left ones.

The init state

A special state, called "init", can be declared to be applied immediately after the instantation of an object, and exactly before the object is returned to the user.

Substantially, it's like setting the state at the end of the last executed init block in the topmost child.

```
class HavingInit

   init
      > "Instance created"
   end

   state init
     function callme(): > "from initial state"
   end

end

h = HavingInit()
a.callme()        // already in init state
```

It is possible to re-enter the init state after having abandoned it via `instance.setState( "init" )`.

Init state and __enter method

The __enter method is in the init state is not useful just to manage transitions re-entering in the init state after having left it. As __enter is called automatically when a state is applied, and the init state is applied to an instance before returning it to the caller, the __enter method in the init state is automatically called after an instance is completely prepared and setup.

This is interesting because it provides a callback that can be set by base classes to provide common post-initialization code for a whole hierarcy.

Consider the following example: instances are automatically stored in a global dictionary under a key that is filled by child classes. Normally, it is necessary to call a virtual function of the base class from the client code after initialization, but whit __enter from init state you can set this behavior in the base class:

```
globdict = [=>]

class Base
   init
      > "Base init"
   end

   state init
      function __enter()
         global globdict

         > "Enter method called for ", self.type
         if self.type notin globdict
            globdict[self.type] = [self]
         else
```

```
            globdict[self.type] += self
         end
      end
   end
end


class TypeA from Base
   type = "Type A"

   init
      > "TypeA init"
   end
end

class TypeB from Base
   type = "Type B"

   init
      > "TypeB init"
   end
end


TypeA()
TypeA()
TypeB()
inspect( globdict)
```

As you can see from the sample, the base initializer is called before the child one, but then the __enter method is applied after child initialization.

Prototype based OOP

Classes and singleton objects have a fixed structure which cannot be changed during runtime. At times, defining the structure of objects dynamically may be useful; for example , property tables may be loaded from a file, or class definitions may be provided externally from another program. Creating an instance is then a matter of copying the structure of an original model item (the prototype), and this operation doesn't bind the structure of the new instance to stay faithful to the base instance definition for its whole lifetime.

We have already seen that arrays can hold both ordinal data and bindings, and these bindings can also be functions. When calling a function bound with an array, the value of self gets defined as the array for which the function has been called. See the following example:

```
vect = [ 'alpha', 'beta', 'gamma' ]
vect.dump = function ()
    for n in [0: self.len()]
       > @"$(n): ", self[n]
    end
end

vect.dump()
```

So, arrays with bindings can be seen as instances of abstract classes, which also hold a ordered set of 1..n values, which are accessible with the square accessors.

Dictionaries can provide another form of prototyped instances. String keys without spaces can be seen as property and method names. To tell Falcon that we'd like to have a dictionary as a prototype, we need just to bless it:

```
function sub_func( value )
   self['prop'] -= value
   return self.prop
```

```
end

dict = bless( [
  'prop' => 0,
  'add' => function ( value )
             self.prop += value
              return self.prop
            end ,
  'sub' => sub_func
   ])
```

As the above example shows, it is possible to place in the dictionary data and functions, either declared directly in the dictionary or elsewhere.

> *Notice the ; after each statement in the internal function declaration. It's necessary because opening a parenthesis suspends the interpretation of end-of-line as statement terminator. Also, notice that after the end of the first function, it is necessary to add a comma, as the program flow returns immediately in parsing the dictionary being formed, and a separation from the next element becomes necessary.*

Blessing (that is, calling the `bless` function on the dictionary) is necessary because dictionaries are meant to hold potentially huge amounts of data (in the order of several hundred thousand items); the non-blessed dictionaries can be distinguished so that applying method on them doesn't require a full scan of their content, but just a search on the standard dictionary methods. Without a blessing mechanism, a simple `len` method applied on the dictionary would have caused first a complete search in all the keys, and then the needed scan in the standard dictionary methods. This is often not desired. Also, blessing a dictionary has the visual effect of declaring it as not just a dictionary.

Blessed dictionaries also can be accessed with the dot accessor, and functions stored in dictionary values and retrieved through the dot accessor are called as methods, with the `self` item correctly set to the owning item. Under every other aspect, they stay normal dictionaries; for example , as shown in the `sub` function, they can still be accessed by their key. Adding new string keys has the effect of adding new properties or methods dynamically; still, it is possible to add any item as a dictionary key, and retrieve it as usual.

```
dict.add( 10 )  // adding 10 to prop
> "test 1: ", dict.prop
dict.sub( 5 )   // and now subtracting 5
> "test 2: ",dict["prop"]  // accessing as a property

// Adding dynamically a new method
dict["mul"] = function( val ); self.prop *= val; end
dict.mul( 3 )
> "test 3: ", dict.prop
```

Except for the fact that blessed dictionaries define their inner data to be accessible also via the dot operator, and while array bindings do not reference the sequential data contained in the owning array, they are mostly interchangeable; so we'll use just blessed dictionaries in the rest of the chapter, eventually specifying specific behavior of dictionaries or array bindings.

Exactly as for class based OOP, properties declared with an underline sign _ are accessible only through the `self` item, becoming private; however, they can be accessed normally through the dictionary interface (they are just strings).

Instance creation

A program relying on prototype OOP usually needs some means to create similar objects (the

instances). This is usually achieved by two means: factory functions or instance cloning.

Prototype factory functions

The most direct way to create an object in prototype OOP is to have it returned from a function. For example :

```
function Account( initialAmount )
    return bless([
      "amount" => initialAmount == nil ? 0 : initialAmount,
      "deposit" => function ( amount ); self.amount += amount; end ,
      "withdraw" => function ( amount )
          if amount > self.amount: raise "Not enough money!"
          self.amount -= amount
      end
      ])
end

acct = Account( 100 )
> "Initial amount ", acct.amount

acct.deposit( 10 )
> "After a small deposit: ", acct.amount
```

Remember that a ; is needed after each statement when declaring functions inside dictionaries.

In this example, Account is a normal function, but it just returns a new instance of the dictionary.

Prototype cloning

The word prototype means that every object can be seen as a prototype of a hierarchy of cloneable and differentiated objects.

Every Falcon item can be cloned through the `clone` method of the base FBOM system. Continuing the above example:

```
nacct = acct.clone()
nacct.withdraw( 100 )
> @ "Left on acct $(acct.amount) and on nact $(nact.amount)."
```

Normally, clone performs a shallow copy of everything that's in the cloned object; this means that other deep objects that may be contained in the instance (as, for example , arrays or other instances) are not cloned themselves. To override the normal cloning process, it is possible to re-define the clone method inside the object:

```
nacct["clone"] = function ( amount )
    newItem = itemCopy(self)
    if amount: newItem.amount = amount
    return newItem
end

acct2 = nacct.clone( 50 )
> "New instance reinitialized with ", acct2.amount
```

Calling the clone will now invoke the function we have written; to avoid using the clone method again (which would cause an endless recursion, we may either store it in a private property or use `itemCopy` function as we did in this example.

Referencing the factory function

A particularly elegant technique is to have the factory function stored somewhere in the returned

vector:

```
function Account( initialAmount )
   return bless([
      "new" => Account,
      "amount" => initialAmount == nil ? 0 : initialAmount,
      "deposit" => function ( amount ); self.amount += amount; end ,
      "withdraw" => function ( amount )
          if amount > self.amount: raise "Not enough money!"
          self.amount -= amount
      end
      ])
end

instance = Account( 10 )
other = instance.new( 20 )
> "Amount in new instance: ", other.amount
```

This doesn't require copying the prototype, which may differ from the base idea of an initial instance as it should be.

A small prototype class sample

Classes themselves can be seen as objects. This is the base idea of reflection in reflective languages as Java and C#. So, it is possible to create classes that will actually have the duty to configure new instances and eventually provide some basic services.

In this example, we create a base class which gives birth to an instance:

```
base = bless([
      "new" => function (prop)
          return bless([
            "class" => self,
            "method" => self["method"],
            "property" => prop
         ]);
      end ,

      "method" => function (); > "Hello from ", self.property; end
   ])

inst = base.new( "me" )
inst.method()

// outputs
Hello from me
```

Notice the usage of the array access operator instead of the dot operator to retrieve `method` during instance creation. Accessing an object via the dot operator, we'd store in the final instance method a reference to the object where the method was declared (that is, our `base` class). Calling `inst.method` would then actually resolve in calling base.method; this is not what we want. The idea is that the final method gets called having the newly created instance as self. By retrieving the method as a simple content of the dictionary, we can propagate it to the child instances, which will see them as functions, and transform them in correct methods as the dot operator is applied.

Operator overloading

As with standard Objects and Classes, prototype object operators can be overloaded. The concept described in Objects and Classes is essentially the same for prototype objects. One only need define the specific operator to overload. Operators fall into the following categories:

- mathematical operators overloading
- comparison overloadin

Mathematical operator overloading

Binary operator overloading take a single parameter (the second operand) and are usually, but not necessarily expected to return a value of the same type as self, or of the same type of the second operand. Binary operators are:

- **add__** Overloads "+" and "+=" operators
- **sub__** Overloads "-" and "-=" operators
- **mul__** Overloads "*" and "*=" operators
- **div__** Overloads "/" and "/=" operators
- **mod__** Overloads "%" and "%=" operators
- **pow__** Overloads "**" operator

```
imBlessed = bless ( [
  'angel' => 100,
  'devil' => 25,
  'add__' => function(val)
    printl('add__ called')
    self.angel += val
    return self
  end,
  'sub__' => function(val)
    printl('sub__ called')
    self.devil -= val
    return self
  end
] )

imBlessed += 25
> imBlessed.angel
aBitLess = imBlessed - 50
> aBitLess.devil

// OUTPUTS:
add__ called
125
sub__ called
-25
```

Unary operators overrides receive no parameters; they are the following:

- **neg__** Overloads the unary prefix negation operator ("-" in front of a symbol).
- **inc__** Overloads the prefix "++" increment operator.
- **dec__** Overloads the prefix "--" decrement operator.
- **incpost__** Overloads the postfix "++" increment operator.
- **decpost__** Overloads the postfix "--" decrement operator.

Little contrived example using `neg__` and `incpost__`

```
unaryExp = bless ( [
  'chg' => 100,
  'incpost__' => function()
    printl('incpost__ called')
    self.chg = self.chg + 1
    return self
  end,
  'neg__' => function()
    printl('neg__ called')
    self.chg *= -1
    return self
  end
```

```
] )

unaryExp++
> unaryExp.chg
tmp = - unaryExp
> tmp.chg

// OUTPUTS:
incpost__ called
101
neg__ called
-101
```

Comparison overload

Comparison operators, namely <, >, <=, >=, == and != all refer to the same overload method: compare. Notice the absence of the "__" suffix. This is both because of historical reasons and because compare doesn't exactly overload operators, but serve a more complex purpose with a different semantic.

The `compare` method is bound to return a number less than zero, zero or greater than zero if the self item is respectively less than, equal to or greater than the comparand item, passed as parameter. Contrarily to mathematical operators, the `compare` method should not raise an error in case the items are not comparable: Falcon VM prefers to have a strategy to sort all the items, even when sorting has no physical reason. When the compare function hasn't any mean to determine a sorting order, it should return nil; this informs the virtual machine that the overload gave up, and that the default ordering algorithm should be applied: items of different kinds are ordered based on the value of their typeId() methods, and items of the same kind are checked based on the place they occupy in memory.

Message Oriented Programming

Message oriented programming (MOP) consists in writing program sections generating and replying to messages (happening now, in the future or even happened in the past) instead of writing direct calls.

Falcon MOP is constituted by three distinct inter-operating entities:

- subscriptions: requests to be notified about events.
- broadcasts: generation of temporary messages.
- assertions: persistent messages that stays in the environment until retracted.

A message is formed by a *name* (also called *event*) and zero or more parameters. The *event* is always and exclusively a string, while any Falcon item can be used as parameters.

As a simple example of MOP, let's rewrite a MOP-oriented printl:

```
subscribe( "printl", {tbp => >tbp} )
broadcast( "printl", "Hello world!" )
```

The `subscribe` call informs the system that we want to reply to *printl* events through the given handler, in our case, the given code block that just prints one parameter. The `broadcast` call sends the parameters (in this case, just "Hello world!") to all the subscribed handlers, if there is any.

Assertions

Falcon messaging model allows to post a single item temporarily or permanently associated with a message.

For example, suppose that some module can be started either before or after some "goodStuff" get readied. We want to complete our work only after we can put our hands on the "goodStuff". So,

```
class MyStuff
   // private data...
   ready = false

   init
      subscribe( "goodStuff", self.configure )
   end

   // more stuff...

   function configure( stuff )
      // good, we have the good stuff
      //... use the stuff to do things...
      self.ready = true
   end
end
```

In another part of the program, the assertion can be posted like this:

```
assert( "goodStuff", "Some stuff to be sent around" )
```

This causes all the already created instances of `MyStuff` to be configured at once, and allows new instances created from now on to be immediately configured.

Listeners can unsubscribe from listening messages and assertions through the `unsubscribe` function, passing the event to which they wish to unsubscribe and themselves.

In example, as configuration is one-time action, the above MyStuff class may wish to unsubscribe once received the message, so the configure method can be rewritten as:

```
   function configure( stuff )
      // ... rest as before
      unsubscribe( "goodStuff", self.configure )
   end
```

It is possible to issue a broadcast with the same name of an existing assertion, so subscriptions to events will respond both to assertions and broadcast on that event. Asserting over a previously existing assertion replaces the previous one with the new data, and also notify the change by re-broadcasting the new value.

To remove an existing assertion use the `retract` function:

```
// no more good stuff this days
retract( "goodStuff" )
```

Retracting a non-existing assertion will raise an error.

Finally, it is possible to query for the current value of an assertion:

```
assert( "goodStuff", "Really good stuff" )
> getAssert( "goodStuff" )
```

Normally, `getAssert` function raises an error if there isn't any assertion active on the required event, but it's also possible to provide a default value that is returned in case the assertion isn't

found, as in the following example:

```
> getAssert( "non-existing", "Ops, we didn't found an assert" )
```

Broadcast Control

Broadcast is performed synchronously. The caller of broadcast waits that the subscribers reply to the broadcast in turn, and returns the value that was returned by the last handler.

The order by which handlers are called is the same order in which they have subscribed. To prevent other handlers to get in control of the message, the subscriber must call the function `consume`, which will grant that after it returns, the broadcast will be interrupted and the `broadcast` function will return its same return value. The following example shows how an appointed subscriber can reply to a message returning a value to the `broadcast` caller.

```
// create a couple of receivers
function f1( target, value )
   if target == "by100"
      consume()
      return value * 100
   end
end

function f2( target, value )
   if target == "by500"
      consume()
      return value * 500
   end
end

// subscribe the two receivers
subscribe( "multiply", f1 )
subscribe( "multiply", f2 )

// multiplying a value depending on the target:
> "Mult 2 by 100: ", broadcast( "multiply", "by100", 2 )
> "Mult 2 by 500: ", broadcast( "multiply", "by500", 2 )

// and a non-existing target...
> "Mult 2 by 1000: ", broadcast( "multiply", "by1000", 2 )
```

A late subscriber can be put on top of the subscribers list by passing an extra `true` value as the last parameter of subscribe. In this way, a message filter coming after other subscribers can prevent the passage of the message to original subscribers by consuming it:

```
function f1(): > "I am f1"
function f2(): > "I am f2"
function f3(): > "I am f3"
function f4(): > "I am f4"

// subscribe regularly f1 and f2
subscribe( "printme", f1 )
subscribe( "printme", f2 )

// but give more priority to f3 and f4
subscribe( "printme", f3, true )
subscribe( "printme", f4, true )

broadcast( "printme" )
```

As seen, the call order becomes {f4, f3, f1, f2}, as the topmost item is f4, inserted with priority on top of f3, they both inserted with priority with respect to f1 and f2.

It is possible to broadcast a message in without waiting for its result creating a *coroutine* ad hoc for

this. We'll explain coroutine in a later chapter, but it's worth to see that, if one doesn't need the broadcast to be in sync with the caller, it can write it as:

```
launch broadcast( "anEvent", ... )
```

Similarly, it is possible to perform a broadcast from a different thread, but threading is argument for another document.

Cooperative broadcast

At times, it's not just useful to "steal" the broadcast signal from the following handlers. It may be also useful to cooperate to form a common result, or to perform different steps of a common work.

Broadcast parameters can be any Falcon item, including arrays, dictionaries and objects. Each participant in the broadcast may alter or manipulate the incoming item(s), as in the following example, where a first subscription step allows participants to ask for a subsequent call.

```
class Subber( id )
   id = id

   init
      subscribe( "process", self.subMe )
   end

   function subMe( requests )
      // will I be taking the second step?
      if random(10) < 5: requests += .[ self.callMe ]
   end

   function callMe()
      > @"Subber $self.id was called back!"
   end
end

// all the subbers
subbers = []
for i in [0:10]: subbers += Subber(i+1)

// action they want us to do
requests = []
broadcast( "process", requests )

// do them
for req in requests: req()
```

The broadcast of the above example gives all the listener the chance to post a request in the vector that is being formed during the process. When the broadcast is complete, the main program executes each posted request.

This allow also for "auction" based broadcast, where the subscribers find an agreement on who should actually process the message as described as the following example shows:

```
class BidToken
   value = 0
   callback = {=> >"No winner" }
end

class Player( id )
   id = id

   init
      subscribe( "play", self.subMe )
   end

   function subMe( tk )
      // tk is a bid token.
```

```
      bet = random(10)
      if bet > tk.value
         tk.value, tk.callback = bet, self.callMe
      end
   end

   function callMe()
      > @"Player $self.id won the auction!"
   end
end

// all the players
players = []
for i in [0:10]: players += Player(i+1)

// action they want us to do
tok = BidToken()
broadcast( "play", tok )
// elect the winner
tok.callback()
```

Automatic marshaling

Objects, instances and blessed dictionaries can be used directly to receive broadcasts, even if they are not callable items. By subscribing a message with an instance, the subscriber indicates that it wants that message marshaled to a function named "on_" + the event name. In example, if an object is suscribed to an event named "bcast", in case "bcast" is broadcast, its "on_bcast" method will be called back.

```
object Processor
   init
      subscribe( "bcast", self )
      subscribe( "evt", self )
   end

   function on_bcast(): > "Received a bcast"
   function on_evt(): > "Received an evt"
end

broadcast( "bcast" )
broadcast( "evt" )
```

In case the subscribed item doesn't provide the needed callbacks, a MessageError is raised. The check is performed at broadcast time because prototype oop constructs may change their structure in the meanwhile, and class instances may change the type of existing properties, making them non callable.

Message Slots

Falcon manages subscriptions, broadcasts and assertions through an object called *Slot*. This object is reflected and accessible from scripts through the `VMSlot` class.

A VMSlot can be accessed through two means: using the `getSlot` function or creating a VMSlot instance. In the first case, if the desired slot doesn't exist (that is, if there aren't any subscriptions or assertions active for that slot), an error is raised; in the second case, the slot is created anyhow, and eventually connected to the *Slot* data if it's not empty.

Operating on VMSlot methods is totally equivalent to using the homologous functions; the only difference is that the *event* parameter is missing, as it's included in the VMSlot object, and any operation on VMSlot is faster as the VM doesn't need to search its internal slot database for the

required *event*.

For example:

```
slot = VMSlot( "event" )

subscribe( "event", function(); > "Received an event"; end )
slot.broadcast()  // == broadcast( "event" )
```

or even:

```
subscribe( "event", function(); > "Received an event"; end )
slot = VMSlot( "event" )  // connects to "event" subscriptions
slot.broadcast()
```

It's possible to create multiple instances of the same VMSlot, even in different modules and in different threads; they all refer to the same internal *Slot* data. So, in case you want to use a VMSlot for faster operation in different unrelated parts of your program, you can simply create a local instance.

Iterating on VMSlots

An interesting characteristic of the VMSlot abstraction is that it is possible to iterate over them in for/in loops. Each loop receives a subscriber in the same order respected by `broadcast`. Using `continue dropping` to delete an item from the collection has the same effect as unsubscribing the handler.

For example:

```
// some subscriber
function f1(): > "I am f1"
function f2(): > "I am f2"
function f3(): > "I am f3"

// subscribe all of them
slot = VMSlot( "bcast" )
dolist( slot.subscribe, [f1,f2,f3] )

// Let's say we don't like f2
for subscriber in slot
    >> subscriber
    if subscriber == f2
       > " (we don't like it)"
       continue dropping
    end
    > " (ok)"
end

// broadcast
>
> "Broadcasting now."
slot.broadcast()
```

The broadcast will activate only `f1` and `f3` as `f2` has been effectively unsubscribed.

> ***Warning**: Better not to use this when the slots can be broadcast from other threads without a proper protection to avoid concurrency on the broadcast list.*

The VMSlot class provides also `first` and `last` methods that return an iterator that can be used to scan and modify the subscription list.

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 understandable and useful groups.

Tables have been the most effective categorization tool for hundreds years, and they still make the fortunes of marketing and strategy gurus. Compared to them, the formal definition of "class" as we know it as the base of object oriented programming is relatively young. Actually, outside IT labs, people use tables to analyze situations and drive decisions. Unsurprisingly, every managerial decision making system involves some sort of grid, or table, that can easily be built with advanced instruments such as pencil & paper, and these tools can drive decisions worth billion of dollars.

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

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

Tables are classes

The base of tabular programming is the Table class, which is absolutely a normal class except for some very special interactions with arrays that are handled transparently. It would be more accurate to say that arrays know what a Table instance is, and are kind with them, rather than seeing the Table class as special .

More precisely, a Table is a class that stores 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 data in rows and column data can be any Falcon object, including other tables, while column names are limited to strings (not necessarily, but preferably, not containing 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 parameters).

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:  Benson   income:   2300
name:   Smith   income:   2000
name:   Jones   income:   2800
name:   Sears   income:   1900
name:    Sams   income:   3200
```

Arrays stored in tables stay available to the outside. For example ;

```
>>> 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 its first element can also be called “name”, while its second element 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 also have bindings; the main difference between the bindings and the table column names is that bindings lay beside the array, while table names lie inside it, and allow indirect access of their ordinal content. This allows virtualiz ing the array 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 the `self` item;

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

and they can also refer 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, other specific of some item types, which can be generally applied to any Falcon item, including numbers, strings and even `nil`.

Default Values

As said, 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| {=> 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 example , 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     { x => x } ] ,
    .[   3000   4999    { x => x *  0.95}  ] ,
    .[   5000   6999    { x => x *  0.93}  ] ,
    .[   7000       0   { 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 code block of the row that applies to an arbitrary price.

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

We can build a more interesting “all in one” function; see the following examples.

```
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 designs 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 tables for data definitions instead of switches, cascades of nested ifs, method overriding 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.

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 we have a set of bank account meta-data, as the interest rate for a certain account category. Then, a table may store all the account types and their interest rates in different pages, which can be marked at different times.

The following class extends the base Table so selecting a page depends 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 calcs.

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

AccTable . setYear( 2008 )
> "Premium account with 1000$ on 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 could also have been easily done with simpler methods, and would definitely also fit OOP; but suppose that it's not just a parameter change in years, but the whole logic that regulates refunds of credit accounts.

Suppose, for example , that starting from year 2007 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'     { x => x * 2.3 / 100}  {x => x * 8.4 / 100 } ]
          .[ 'business'  { x => x * 0.8 / 100}  {x => x * 4.2 / 100 } ]
          .[ 'premium'   { x => x * 3.2 / 100}  {x => x * 5.3 / 100 } ]
          ] )

      // in year 2008
      self.insertPage( nil , .[
          .[ 'basic'    {x => x * 3.2 / 100}  {x => x * 7.6 / 100 + 50} ]
          .[ 'business' {x => x * 1.1 / 100}  {x => x * 4.5 / 100 + 50} ]
          .[ 'premium'  {x => x * 3.4 / 100}  {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$ on 2007 costed: " , \
   AccTable.creditCost( 'premium' , 1000 )

AccTable.setYear( 2008 )
> "Asking for 1000$ on 2007 costed: " , \
   AccTable.creditCost( 'premium' , 1000 )
```

The output of this program is:

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

Doing the same thing with traditional OOP constructs would require to program this variability since the very beginning of the project, or face the eventuality that the classes initially designed to represent bank accounts will be missing just the last sparkle of flexibility needed to implement the last twist that the marketing guys have thought of 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 just isn't enough, and thinking of some problem category as two dimensional tables (or eventually as three-dimensional tables-in-time as in this example) fits much better the more destructured and pragmatic approach to problem definition and analysis that business professionals are accustomed to use (and send down to the development departments for implementation).

Additionally, tables have some further utility worthy of consideration.

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 if it doesn't win. After each bidding, the bid amount is removed from a pool of resources initially given to each bidder.

The first two algorithms haven't 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. We'll have then a state property, called `storage` , where the bid is placed, and a property filled by the calling

table indicating if the algorithm was winning 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 correct creation of 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 all the 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 play the trick.

```
      bidders = self.getPage()
      arraySort( bidders, { 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. We'll need then an extra column where to store a method doing 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 normally 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 much simpler:

```
    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 setting here the value for the winner, we use this method which does the same 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 into

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

And now it's possible to remove the extra column “genbid”.

Iterators

Iterators are objects meant to access sequentially other structures. They are the preferred way to partially scan a long sequence of data. They provide all the functionalities of a for-in loop and they operate on the same structures that the for/in loop manages, but they can also be used to scan a sequence backwards, insert data at a certain position or record one or more positions for further usage.

Iterators can get invalidated because of operations on the underlying sequence, or because they are moved outside the sequence range. The rules for invalidation varies depending on the underlying structure, but usually adding or removing an element may cause invalidation. The only safe way to

change a structure so that the iterators stay valid is doing that through the iterators themselves.

An iterator is created calling the `first()` or `last()` method on sequences, or using the constructor of the `Iterator` class. For example :

```
iter = Iterator( [ first, second, third ] )
vector = [ "a","b","c" ];  iter = vector.first()
iter = vector.last()
```

Are all valid ways to create an iterator on an array. In case of random access sequences (the items you can access using an integer index in square brackets), the Iterator class constructor may be given an optional numeric parameter indicating the initial position where the iterator is placed. The integer accepts array convention where a negative number indicates a position from the end of the sequence. Iterators for other sequences accepts only 0 or -1 (for the last element). Iterators over attributes (which scans the objects that have been currently assigned a certain attribute) can be created only starting from the begin of the sequence.

Iterators provide the `value()` to read or set the current value. Using it on an invalid iterator raises an access error; the `hasCurrent()` method returns true if the iterator is valid, so it can be used to check if the iterator has been moved outside the sequence.

The `next()` and `prev()` methods move the iterator respectively forward and backward in the sequence. They return true if there is a next or previous element, and false if the operation caused the iterator to move outside the bounds of the sequence, making it invalid. Since when that happens it is often too late, `hasNext()` and `hasPrev()` methods are also provided to allow last element special processing. An element that doesn't have a next element is the last element of a sequence, and if it has no previous element it is the first.

So, we can rewrite a common have a nice day! for/in loop like the following:

```
list = List( "have", "a", "nice", "day" )
iter = list.first()

while iter.hasCurrent()
   >> iter.value()
    if iter.hasNext()
      >> " "
    else
      > "!"
    end

   iter.next()  // still works if hasNext() is false
end
```

For dictionaries, the `key()` method retrieves the current element key. For example , the following loop we change every item in the dictionary if the key starts with c.

```
dict = [ "alpha" => 1, "beta" => "2", "charlie" => 3,
         "carl" => 4, "day" =>5 ]
iter = dict.first()

 while iter.hasNext()
    if iter.key()[0] == "c"
      iter.value( "Changed" )
    end
    iter.next()
end

inspect( dict )
```

It is possible to partially scan a dictionary, which is ordered by key, using the dictBest() function. It returns an iterator to the item considered lexicographically equal or immediately greater than the

searched key. The above example can be rewritten using dictBest like this:

```
dict = [ "alpha" => 1, "beta" => "2", "charlie" => 3,
         "carl" => 4, "day" =>5 ]
iter = dictBest( dict, "c" )

while iter.hasNext() and iter.key()[0] = "c"
   iter.value( "Changed" )
   iter.next()
end

inspect( dict )
```

Iterators can be compared for equality; iterators are equal when they point to the same item in the same collection. For example , this loop would work too:

```
list = List( 1, 2, 3, 4, 5, 6, 7 )
iter = list.first()
lastIter = list.last()

loop
   > "Element: ", iter.value()
   if iter == lastIter: break
   iter.next()
end

inspect( iter )
```

Iterators can be used to remove or insert an item in a collection. In the case of dictionaries, both the key and the value must be provided; if the iterator is already pointing to a position which is lexicographically correct for the given item, then the dictionary is not scanned for a correct position before insertion. Compare the following code:

```
dict = [ "alpha" => 1, "beta" => 2, "charlie" => "3" ]
 if "beta" in dict
   dict[ "bravo" ] = 4
end
```

With this:

```
dict = [ "alpha" => 1, "beta" => 2, "charlie" => "3" ]
iBest = dictFind( "beta" )
 if iBest
   iBest.next()
   iBest.insert( "bravo", 4 )
end
```

In the first case, two scans have to be performed on the dictionary; the first to search for the value, the second to insert. In the second case, Falcon will use the information in the iBest iterator to avoid the second scan. However, notice that creating an iterator is a relatively heavy operation, and usually a single dictionary search is faster, so this technique is better exploited with repeated insertions.

List Comprehension

In a realm halfway between the functional and the procedural language paradigms, there's an hybrid region made of lists, and built up on the way you can process them. The king of that reign is the "list comprehension", that is, a formal definition of the contents of a list.

More technically, according to wikipedia:

> *A list comprehension is a syntactic construct available in some programming languages for creating a list based on existing lists. It follows the form of the mathematical set-builder notation (set comprehension) as distinct from the use of map and filter functions.*

And more practically, a list comprehension is a compact construct that can be used to create a set starting from another set. In this, it is similar to the `map` functional operator, but it has two practical advantages: first, the target set needs not to be of the same nature of the original set, and second, it is possible to explicitly specify a source that is not exactly a set, but that can generate items.

Falcon model of list comprehension is a bit more advanced with respect to the base definition, and allows to specify also the nature of the target set. Actually, comprehension in Falcon is performed through the `comp()` method of *sequence* classes and items.

Items offering the sequence interface are currently:

- Arrays
- Dictionaries
- Lists
- Sets

Comprehension components

In Falcon, a comprehension is composed of three elements:

```
target_set.comp( generator, filter )
```

The **target_set** is the sequence that will receive the elements of the comprehension. It may be empty, but needs not to be. It may be an unordered list (an array, or a List class instance), or constraints its member somehow (as a Set instance, or as a Dictionary).

The **generator** is a sequence, a range or a generator function returning one item at a time. As a sequence, it can be anything that can be accessed through an iterator.

The **filter** is an optional element that can provide:

- A predicate stating which elements can and cannot be part of the target set
- A modified copy of the items provided by the generator.

In short, the filter is a function (or in general any callable item) that receives two parameters: the item being currently extracted from the generator and the forming set (that is, the **target_set** itself). If present, its return values will be added to the target set, unless it returns an Out of Band "1", in which case, the elements will be discarded. The filter may also terminate the comprehension returning an out of band 0 integer value.

The `comp()` method returns the sequence itself, so it is possible to assign the result of a comprehension to a target variable directly as it is formed.

Basic examples

Array of pair values from 2 to 10 (included):

```
pairs_in_10 = [].comp( [2:11:2] )
```

Same, as a list:

```
pairs_in_10 = List().comp( [2:11:2] )
```

Sorting an unordered list of elements:

```
sorted = Set().comp( ["oranges", "apples", "peaches", "bananas", "grapes" ] )
for item in sorted: > item
```

Lowercase version of an uppercase list:

```
lcase = [].comp( .[ "A" "b" "Cc" "DDD" ], { v => v.lower() } )
```

Accepting values below 10:

```
less_than_10 = [].comp( .[ 1 5 18 3 9 12 15], { v => v < 10 ? v : oob(1)} )
```

Here, oob(1) asks `comp()` list to discard this entry.

Warning about duplicate entries (notice that we receive the *set* as the second parameter of the filter function):

```
ordered = Set().comp( .[ "N" "C" "N" "D" ], { v, set =>
   if set.contains(v)
      > "Duplicated: ", v
      return oob(1)
   end
   return v
   } )

for item in ordered: > item
```

Accepting the first 10 random values (and ordering them):

```
randomSet = Set().comp( random, { v, set => set.len() == 10 ? oob(0): int(v*100)} )
for item in randomSet: > item
```

Dictionary comprehension

To form a dictionary comprehension, each item received must be a "pair", that is, a two elements array. For example:

```
dict = [=>].comp( .[ .[ 'a' 1] .['d' 2] .['c' 3] .['b' 4] ] )
for k,v in dict: > @"$k = $v"
```

The filter function can be used to turn a sequence with one element in a dictionary. The following example creats a dictionary where each uppercase letter is associated with its UNICODE value.

```
dict = [=>].comp( [0:26], {v=> ['A'/v, ord('A')+v]} )
```

Generators

A useful extension of the comprehension system is provided by the generators. A generator is simply a function returning an item at a time (or a pair of item in case the comprehension is dictionary-wise). When the generator declares it has terminated by returning an out of band 0 integer.

For example, the following generator returns a random count of random numbers:

```
function randrand()
   // terminate 1/10th of times
   if random() > 0.9: return oob(0)
   // otherwise, return a random number
   return random( 1, 10 )
end

rvals = [].comp( randrand )
inspect( rvals )
```

Using a closure may be more interesting; this creates a lowercase alphabet:

```
function makeAlphabet()
   m = 'a'
   return function()
      if m > 'z': return ^+ 0
      k = m
      m /= 1
      return k
   end
end

abet = [].comp( makeAlphabet() )
```

Notice that we used the `^+` *mark out of band* operator, which is faster and more compact than `oob(0)`, and notice also the UNICODE string increment `/=`. Finally, notice that `makeAlphabet` must be called to provide `comp()` with the new closure it returns.

Another interesting usage of generators involves callable objects, or functors. We can write the above example as:

```
object doAlpha
  m = 'a'

  function call__()
      if self.m > 'z': return ^+ 0
      k = self.m
      self.m /= 1
      return k
   end
end

abet = [].comp( doAlpha )
```

Closures are generally more compact and efficient than functors, but functors are more flexible and they may even alter their behavior in during the comprehension. In fact, we may provide a functor method as the filter, so that...

```
class randgen( count )
  count = count

  function call__()
     return random(1,10)
  end

  function filter( elem, set )
     if set.len() >= self.count: return ^+ 0
     return elem
  end
end

r = randgen(10)   // max 10 elements

rlist = [].comp( r, r.filter )
```

In the above example, the instance `r` is used both as a functor to generate the items, and as a filter to determine when the sequence should be considered complete.

Completion comprehensions

As said in the introduction, a comprehension needs not to be performed on an empty sequence. Actually, the items created in the comprehension are added (appended) to the target sequence; for example:

```
tgt = [1,2,3]
tgt.comp( [0:4], {v=> 'A'/v} )
inspect( tgt )
```

As seen, the `tgt` array is modified on place, receiving the items generated by the comprehension.

Custom comprehensions

The `comp` method can also be applied as an FBOM method to objects and blessed dictionaries exposing a method named `append`. The method will be called (atomically) for each item generated by the comprehension (after it has been accepted by the filter).

For example, the following code fills an array in an object property:

```
class MetaArray
  _array = []

  function append( data )
     > "Appending: ", data
     self._array.add( data )
  end

  function display()
     > self._array.describe()
  end
end

ma = MetaArray()
ma.comp( [0:3], {x=> x*2} )
ma.display()
```

The same code can be expressed as a blessed dictionary:

```
d = bless([
  "array" => [],
  "append" => function( data )
     > "Appending: ", data
     self.array.add( data )
  end
])

d.comp( [0:3], {x=> x*2} )
inspect( d.array )
```

In this case, the "append" method will take ownership of the comprehension process. In fact, blessed dictionaries cannot normally receive comprehension items as normal dictionaries, unless filtered through an `append` method.

For/in generators

The for/in loop provides a procedural version of the list comprehension construct. Generators can be used as the source element of the for/in loop, as in the following example:

```
function makeAlphabet()
   m = 'a'
```

```
   return function()
      if m > 'z': return ^+ 0
      k = m
      m /= 1
      return k
   end
end

f = makeAlphabet()

for i in f
   forfirst: >> "Alphabeth: "
   >> i
   formiddle: >> ", "
   forlast: > "."
end
```

Error recovery

You'll remember our first interactive Falcon script: that was the one asking you for your age and then entering a loop congratulating with you many times for your past birthdays.

The int() function tried to convert a string (what you typed) into an integer (your age), but if this was not possible for some reason, a runtime error appeared instead, and the program was terminated. Here follows a reduced version of that script that will serve our needs:

```
print( "Enter your age: > " )
age = int( input() )

count = 0
while count < age
   count += 1
   printl( "Happy belated birthday for your ", count, "." )
end
```

Falcon provides a mechanism to handle unexpected situations that may arise in a program. Many library functions and language constructs use this mechanism to communicate with the controlling script about unexpected situations, but this system is also available to the script itself, so that script writers can take advantage of this. It's called exception raising .

Every time the Virtual Machine, one of the library functions or even other script parts run into a potentially dangerous situation, they raise an exception. If this exception is not handled somehow by the script, it is handed back to the system; the Falcon interpreter will print an error message and exit.

> *If the Falcon Virtual Machine is used by an embedding application to run some scripts, the embedder has the ability to set a top level exception handler. This will usually grant the embedding application the ability to know about fatal errors in the scripts, and take sensible actions (as i.e. mailing the administrators).*

There are a set of exceptions that are called unstoppable . These exceptions are raised by library functions or by the Virtual Machine itself if it finds some critical condition that may prevent scripts from working, as for example script bytecode corruptions. In those situations, letting the scripts intercept the exceptions would not be wise, hence the need of unstoppable exceptions.

Exceptions can be handled by the script by using the try - catch control block:

```
try
   [try statements]
[ catch [object_type] [ in error_variable] ]
```

```
   [ catch statements ]
end
```

Each catch block can intercept a certain kind of variable. The working principle is the same as the `select` statement; a type can be one of the type names, or it can be the name of a symbol declared somewhere in the program.

Try-catch blocks can be nested (put one into another) or combined with any other Falcon block statement (`if`, `while`, `for`, `function` and so on). The try-catch block functionality is as follows: whenever an instruction inside the try (try-statements) causes an exception to be raised, the control flow is immediately broken. If a catch block is present, the type of the raised object is matched against the type specifiers of the catch blocks. Overall types (as i.e. StringType or ObjectType) get precedence, then the specific symbols used as specifiers are considered in the order they are declared in the catch clauses. For this reason, catch blocks intercepting subclasses should be declared before the ones intercepting parent classes. Finally, if none of the typed catch blocks matches the raised exception, the raised error is passed to a catch hander without type declaration, if present. If a typeless catch clause is not present, the error is then raised to the application level and this usually terminates the script.

The following example ensures that the user will write a numeric entry:

```
age = 0
while age == 0
   print( "Enter your age: > " )

   try
      age = int( input() )
   catch
      printl( "Please, enter a numeric value" )
   end
end
```

A catch clause may have an optional variable that will be filled with the exception that has been raised in the try block. The exception can be any Falcon item (including numbers, strings and objects) that describes what exactly was the error condition. By convention, the Virtual Machine and all the library functions will only raise an object of class `Error` , or one of its subclasses. However, scripts and other extensions libraries may raise any kind of item.

The Error class provides a series of accessors, that is, methods that are specifically used to access data in the inner object. Normally, scripts are not very interested in peeking the data inside an Error instance; usually, the embedding application is the entity that is meant to intercept errors and deal with them. For this reason, the embedding API puts at library disposal a C++ class called Falcon::Error; in case the script wants to intercept it, and only in that case, the C++ object is wrapped in Falcon object, and methods are used to query the internal Falcon::Error C++ instance. This is because intercepting and analyzing Error instances from scripts is considered an extraordinary operation; the overhead introduced by using methods instead of plain properties to retrieve Error values is marginal with respect to the advantage the embedding application receives by being able to use directly C++ objects in its code when a forbidding error condition is encountered by the script.

The content of an Error Objects is enumerated in the Function Reference manual. Please, refer to that guide for the details.

Now we can print a more descriptive error message about what the user should do in our test program:

```
age = 0
while age == 0
```

```
   print( "Enter your age: > " )

   try
      age = int( input() )
   catch in error   // any variable name is ok here
      printl( "Oops, you caused the error number ", error.getCode(),
            "\nwhich means that: ", error.getMessage() )
      printl( "Please, enter a numeric value" )
   end
end
```

Do not confuse the `Error` class with the above `error` variable: Falcon is fully case-sensitive, so the variable we named `error` in the above code is just a normal variable receiving an `Error` class instance.

Notice that the catch block is not immune to error raising. If an exception is raised inside a catch block, it will have exactly the same effect as if it were raised in any other part of the program: it may be caught again with another try/catch block, or it may be left to handle to the above handlers, or finally to the Virtual Machine. We'll see in a moment how this fact can be useful.

The `try` instruction can be abbreviated with the : operator; it wont be possible to catch any error in this case, but this may be useful in case any possible error must simply be discarded:

```
    try
       age = int( input() )
    end

   // is equivalent to

    try: age = int( input() )
```

Raising errors

It is interesting to be able to raise errors; the execution flow is immediately interrupted and a possible error manager is invoked, so raising errors inside the scripts may often obviate the need for "if" sequences, each of them checking for the right things to be done at each step. The keyword `raise` makes an item to be thrown and treats it as an exception.

The script may choose two different approaches to raise errors: one is that of creating an instance of the Error class using the Error() constructor, which accepts the following parameters:

```
Error( code, message, comment )
```

However, sometimes it is useful to throw a lighter object. Suppose that we want to set a maximum and minimum age in our example, and that we cause an error to be raised when those limits are not respected. In this case, that we may call flow control exception raising , having a full error to be raised may be an overkill. Follow this example:

```
age = 0
loop
   print( "Enter your age: > " )

   try
      age = int( input() )
      if age < 3: raise "Sorry, you are too young to type."
      if age > 150: raise "Sorry, age limit for humans is 150."

   catch StringType in error
      printl( error )
      // age has been correctly assigned. Change it:
      age = 0
```

```
   catch Error in error
      // it's a standard error of Error class, manage it normally
      printl( "Oops, you caused the error number ", error.code,
            "\nwhich means that: ", error.description )
      printl( "Please, enter a numeric value" )

   catch in error
       printl( "Something else was raised... but I don't know what..." )
       printl( "So I raise it again and the app will die." )
        raise error
    end
end age != 0
```

In this way, we have a controlled interruption of the normal code flow which is passed to the StringType catch branch, with a minimal overhead with respect to the equivalent code performed with a series of branches. If the weight of those branches becomes relevant, the exception code flow control may be even more efficient (the virtual machine management of try-catch blocks is comparatively light with respect to any other kind of operation), while it may be more elegant, and possibly more readable.

It is also to be noticed that the caught variable may be parsed through a select statement. This may or may be an interesting opportunity, depending on the needed flexibility. The above code is equivalent to the following:

```
   // the rest as before...

   try
      age = int( input() )
      if age < 3: raise "Sorry you are too young to type."
      if age > 150: raise "Sorry, age limit for humans is 150."

   catch in error
      select error
          case StringType
            // manage strings as before

          case Error
            // manage Error instances as before...

          default
            // print something as before...
            raise error
      end
   end
```

This solution is a visually a bit less compact, requiring three indent levels where the previous only needed one. Also, the VM has an opcode that manages a typed catch a bit faster than a select statement (it's one VM opcode less, actually, but the opcode that is skipped with the typed catch approach is quite fast to be executed). However, it presents two advantages: first of all, it is possible to execute some common code before or/and after any specific error management. Secondly, the select code may be delegated to a function (or to a lambda) that may be changed on the fly during program execution, actually changing the error management policy for that section. Through this kind of semantics, a common error management policy may be given to different handlers. As this doesn't prevent writing specific typed catches, each error management code may be highly customized through a combination of static typed catch statements and dynamic catch-everything statements passing the raised value to a common manager.

Falcon modules

Falcon is a modular language by design. It is provided with a Virtual Machine oriented runtime

linker that is able to fulfill script requests about base module loading. By default, the scripts are provided with the Core module, a set of functions, classes and objects that are somehow part of the language; for example, it contains the `typeOf`, `int`, and `len` functions, the `Error` class and its children, and so on. The core module is always present in Falcon, although embedding applications may decide to override it.

It is then possible to create binary and falcon language modules that can be loaded by final scripts. Explaining how to create binary modules is beyond the scope of this manual; here we'll see how to create Falcon language modules.

The export directive

Every symbol defined in a module is private to that module, and cannot be referenced elsewhere, unless it is explicitly exported with the `export` keyword. The export keyword can be placed everywhere in the file, and has this grammar:

```
export symbol_name [ , symbol_name, ..., symbol_name ]
```

Many `export` statements may be present in one file, so that using a list of symbols in a line or using several `export` statements has the same effect. If used without any symbol name, `export` will have the effect of exporting all the symbols in a script (and other export statements will be signaled as errors).

The load directive

The load directive instructs the Module Loader that the current script would like to have other scripts loaded as well. This is the format of the load directive:

```
load module_logical_name
```

The "logical name" of a module is handed to the module loader, that will try to resolve the module name so to find it based on the following rules:

When a module is searched in a path, first a binary module that match the logical name is searched; then the loader will search for a pre-compiled module (a binary file with the same name of the logical module name, and the .fam extension). Finally, a source .fal script will be eventually loaded and compiled on the fly.

Partitioning

Partitions are logical (and possibly physical) subdivisions or categories in which modules are organized. A partitioned module resides in a sub-portion of the logical space in which the modules reside. A partition usually is physically represented by a folder, a directory, a link or a file system or a data block in a compressed file. However, partitioning need not necessarily be physical; it may also be a different set of modules provided by an embedding application. The standard Falcon loader, used by the command line interpreter, uses subdirectories in media declared in `FALCON_LOAD_PATH` as partitions.

Partitions are indicated in the `load` directive (and subsequently in the module name) as dot-separated symbols.

For example, the following directive:

```
load networking.http
```

will search the module "http" in a partition (subdirectory) called "networking" in locations indicated by the load path.

> *The `falcon` command line interpreter will automatically add the path of the main module being currently executed in front of its module search path. This behavior is considered "standard", and compliant embedding applications will maintain it. So, scripts, especially standalone ones, can safely assume that the location from which they were loaded will be the first searched for required modules and partitions.*

Partitions can be nested. Consider the following directory tree:

```
main.fal

data/
   calendar.fal

engine/
   mengine.fal
   utils/
      smaller.fal
      larger.fal
```

The main script may load all the modules through the following directives:

```
load data.calendar
load engine.mengine
load engine.utils.smaller
load engine.utils.larger
```

The load directive, and the module logical names, are always relative to the topmost location (or locations) indicated by the application load path. For example, if it is the engine that needs to load the modules in the utils partition, it will need to repeat all the path. Supposing, that the above "larger" module needed to load the smaller one, it would have needed to declare:

```
load engine.utils.smaller
```

which is the complete name under which the "smaller" module is known in its application instance. However, sibling modules and submodules are also known, and those concepts can be used to simplify load directives, as indicated in the following paragraphs.

Sibling modules

Modules may know that they are part of a logical partition, and they may be willing to rely on other modules being in the same partition, or in sub-partitions of it. Starting a load directive with a dot, a module declares that it is searching for a module in its same partition. For example, the above larger module that was loading a sibling smaller module may have done that using the following syntax:

```
load .smaller
```

This notation can be used for sibling partitions. In the above example, the mengine module may load the modules in utils partition through the following directives:

```
load .utils.smaller
load .utils.larger
```

Submodules

Submodules are modules that are logically subordinate to the current module. It is common practice to put submodules in a partition named after the module logical name. For example, suppose that an engine module requires two submodules part_a and part_b to run. If those elements are just logical subdivisions of the engine module itself, they may be physically ordered in a filesystem after the following scheme:

```
engine.fal
engine/
   part_a.fal
   part_b.fal
```

In this case, the keyword self may be used as the root of the partitioning used in the load directive. That will instruct the module loader to search in sibling partitions with the same name as the calling module:

```
// we are inside engine.fal

load self.part_a
load self.part_b
```

This is equivalent to

```
// we are inside engine.fal

load .engine.part_a
load .engine.part_b
```

But using the self keyword to load submodules has the double advantage not to require the script writer to know the name under which the module will be known when the work will be released, and that to immediately identify the loaded modules as submodules, logically dependent from the owner loading them.

Direct name loading

The load directive may also contain a string pointing directly to a .fal script, .fam module or binary loadable module. The load path may be relative to the current script location or to the top of the load path, or absolute in case it starts with a "/". For example:

```
load "submods/mymod.fal"
load "/usr/share/falcon/amodule.fal"
```

The first entry will load the file called mymod.fal in a "submods" directory that will be searched through the script load path, while the second one will load the module in a globally visible directory.

The Slave/Master test

A minimal test demonstrates module loading abilities. This is a typical loadable module that exports some symbols:

```
function slave_init()
   global shared
   shared = "Original"
   printl( "SLAVE - Shared init: ", shared )
end

function slave_func( param )
```

```
   print( "SLAVE - parameters: " )
   for elem in param
      print( elem, ", " )
   forlast
      printl( elem, "." )
   end
   return "Slave is done."
end

function slave_check_shared()
   printl( "SLAVE - shared data is now: ", shared )
end

export slave_func, shared, slave_init, slave_check_shared
```

And this is a typical script loading a module.

```
load slave

slave_init()
// "shared" is imported from slave
printl("MASTER - a shared variable: ", shared )

// now we pass some data to the slave routine
elem = ["A", "list", "of", "strings" ]
retval = slave_func( elem )
printl( "MASTER - return from slave: ", retval )

// and we force the slave to use our data
shared = "Changed from master"
slave_check_shared()

printl( "Done." )
```

Save the first code piece as "slave.fal", and the second as "master.fal". Now you can proceed in two ways:

It is possible to compile the slave script into a module .fam and then launch the master script:

```
my-computer$ falcon -c slave.fal -o slave.fam
my-computer$ falcon master.fal
```

or launch directly the master script. The falcon command line interface will search for sources with matching names and will try to compile them on the fly.

Of course, loaded modules can request in turn load other modules; however, a loaded module is not exactly "owner" of the modules it loads. The load directive is just a "pretty please" said to the Falcon enabled application to provide the required modules before starting the script. The embedding application may ignore the request or provide its own substitute images, or it may even load the required module and change some of the items and functions with its own.

The link step is made so that when a module is loaded in to the Virtual Machine, every symbol it needs has been already exported by some other module. If this doesn't happen, the virtual machine will issue a link-time error to the controlling application, and the script won't be executed.

Module initialization code

Since version 0.8.12, the Falcon virtual machine executes the main code of a module as soon as it is linked (that is, loaded and included in the VM). In this way it is possible to create initialization code which will be executed before the module can be known and used by others.

For example, suppose that a "configuration" module wants to export a list of items to work on. It is now possible to write a very simple Falcon module exporting just a vector of items to be managed:

```
// This is the configuration.fal module
configuration = "Item A", "Item B", "Item C", "Item D"
export
```

And a user module will just need to load the configuration and use it:

```
// This is the main.fal module
load configuration
for element in configuration
   > "Working on ", element, "..."
   // ...
end
```

To know if the module is the topmost module of a load hierarcy, that is, to know if a module is currently used as "main" module, starting and executing a complete falcon program, the `vmIsMain()` function is provided.

This function returns true if the module in which is called has been directly launched by the command line `falcon` command, or by an embedding application; otherwise it returns false. This can be used to execute some code in the main part of the script only when the module is loaded directly. For example, service modules providing functions and objects to be used by others may also provide a main section that is used for testing; if the service module is called directly, vmIsMain() returns true and some testing code can be performed:

```
// This will always be executed when this module is linked in the VM
configuration = "Item A", "Item B", "Item C", "Item D"

if vmIsMain()
   // this will be executed only if loading via "$ falcon configuration.fal"
   > "Testing contents of the configuration..."
   inspect( configuration )
end

export
```

Now, our `main.fal` module will work as previously, but executing directly `configuration.fal` will cause the contents of the configuration to be inspected.

However, init blocks of objects are executed right after the link step and right before the Virtual Machine proceeds, allowing the calling application to link another module. So, if you need some initialization for a module before its main code has a chance to be executed, it is possible to create an object with an init block at the sole scope of initializing the module.

Implicit and explicit import

Falcon compiler believes that everything that has been seen in the main body of a script without being formerly assigned a value is to be found in another module. See this simple script:

```
a_var = 0
printl( a_var )
```

The variable `a_var` is assigned a value, and so Falcon decides that `a_var` will be a symbol owned by the module where it is assigned. On the other hand, `printl` symbol has not been given a value; when it's first met it has not been assigned, so the compiler supposes that it must be externally provided. The Virtual Machine will check this supposition at link time by searching the exported symbols for printl; as it is found in the Runtime module (loaded and linked automatically by the command line tool), the deal is done and the script can proceed.

As the item holding `printl` symbol is shared among all the modules, changing this item in a

module will cause all the modules to see this change immediately, and to start using the new function instead of `printl` . So:

```
a_var = 0
printl( a_var )
// from now on, printl will be remapped to print
printl = print
printl( a_var, "\n" ) // will actually call print, so we add a newline
```

This is a powerful feature, but as any powerful feature it must be used cautiously. In the next example, things don't go straight:

```
a_var = 0
// from now on, printl will be remapped to print
printl = print
printl( a_var, "\n" ) // will actually call print, so we add a newline
```

What has been changed? The first reference to `printl` is now an assignment, and so the compiler decides that we want `printl` to be a symbol private for our module. Asking to export it wouldn't work (actually, the Virtual Machine would raise an error for double definition of a shared symbol). How can we tell the compiler that we want `printl` to be the same item that has been imported by all the other modules?

By using the explicit `import` directive:

```
// some part high in the module
 import printl

// after much code

// from now on, printl will be remapped to print
printl = print
printl( "\n" ) // will actually call print, so we add a newline
```

The first naming of `printl` evaluates in an auto-expression that seeks for `printl` value. The compiler will optimize it anyway, but it will record the fact that the module is seeking for `printl` elsewhere. In general, you may import explicitly a set of symbols from the environment by declaring them in an array that is never assigned:

```
// some part high in the module

import shared, printl
...
shared = "Value from master module."
```

In this way, even if the master modules assigns a value to the `shared` variable, this won't turn the variable into a module private declaration, as the module explicitly seeks it elsewhere.

Local import and namespaces

The load and export directives are adequate to build monolithic applications which are broken in sub-modules for convenient storage of strongly related elements. However, when it is necessary to access symbols provided by foreign libraries providing utility functions and classes, it is better to name the symbols after the library that provides them, or eventually to chose a different name to indicate those symbols.

In fact, two libraries providing similar functions and not knowing each other may export symbols having the same name, and this would result in a name clash that would cause the Falcon Virtual Machine to raise a link-time error.

To avoid this problem the import/from directive (also called local import) is provided. Local import stores the symbols that the program is willing to access into a *namespace* where they can be safely accessed, forcing the Virtual Machine to ignore any export request coming from the loaded module.

The import/from directive has the following grammar:

```
import [sym1, sym2, ... symN] from <modname> [in namespace|as alias]
```

The `modname` specifying the module name where the symbols are loaded from has the same format of the module name used by the `load` directive. It can indicate sibling, child or even top-level modules; a relative or absolute path specifier between quotes may also be specified.

For example:

```
import  func0  from  topLevelMod
import  func1  from  .sibling
import  func2  from    self  .child.subchild
import  func3  from  "/usr/share/falcon/utils.so"
```

To form the namespace containing the desired symbols, the leading "self." and "." in the module names are removed, and path separators are turned into "."; so, to access the above symbols, the following code can be used:

```
topLevelMod.func0()
sibling.func1()
child.subchild.func2()
usr.share.falcon.func3()
```

The local import system also performs name checking at compile time; accessing an unknown symbol from a namespace declared through import/from will raise a compile-time error.

> *We're using only functions in this examples for brevity, but any global symbol can be used in the import/from clauses; this includes classes, objects and global variables.*

However, if the symbols to be imported from a module are too many or are not known in advance, (i.e. because created dynamically by a code generator), it is possible to request a generic local import by not specifying any import symbol. The compiler will then generate a request to import any symbol accessed in read-mode in that namespace, and name mismatches will be detected at link time by the Virtual Machine:

```
import from someMod

someMod.func0()
someMod.func1()
...
someMod.funcN()
```

It is possible to use an arbitrary name instead of the module name as the namespace for the loaded symbols by specifying an alias after the "`in`" keyword; in this way it is also possible to specify different aliases for the same module. For example :

```
import funcA from "/usr/share/falcon/utils.so" in utilsA
import funcB from "/usr/share/falcon/utils.so" in utilsB

utilsA.funcA()
utilsB.funcB()
```

Notice that while it is not possible to locally import a symbol directly in the main global namespace visible from a module, that same symbol may be assigned to a global variable and used directly. For example :

```
import func from "/usr/share/falcon/utils.so"

func = usr.share.falcon.utils.func
func()
```

If all the symbols to be imported from a module do n't fit gracefully on a line, it is possible to specify more import/from directives referencing the same module (and eventually the same alias) declaring different symbols. For example :

```
import func, func2 from mod1
import func3 from mod1

import func4 from mod1 in mod
import func5 from mod1 in mod
```

Finally, it is possible to seamlessly merge `import/from` and load directives in the same program, even referencing the same module. Using load will just ask the virtual machine to honor the export requests of the target module and to make globally visible the exported symbols, while `import/from` actively searches for symbols inside the target module, ignoring exported symbols. If a module is linked just because of `import/from` requests, the virtual machine won't honor its exports, but if there is at least one load request, then exports will be fulfilled and made available to any module in its main namespace.

> *Symbols declared with a leading "_" are considered private of the declaring module, and they won't be exported through `export all` requests nor be visible in `import/from` requests.*

Local import in global namespace

In Falcon, it is possible to assign an imported symbol (with eventually its own namespace) to a local variable, and use that one instead, like in this example:

```
import func from module
myFunc = module.func
...
myFunc()
```

this requires the Virtual Machine to execute the code in the module, as the assignment is an explicit VM operation. It is possible to instruct the VM to link the required foreign imported symbol into a local symbol in the global namespace, using `as` instead of `in`:

```
import func from module as myFunc
..
myFunc()  // actually, it is an alias for func in the given module
```

In this way, it is possible to bypass the standard namespace assignment in explicit import; just, name the local alias after the original name:

```
import func from module as func
...
func()  // a private, safe copy of func in module
```

The advantages of doing this instead of using the `load` directive are:

The `load` directive is meant to pile up and build an application made of several components that have been divided into modules to be more handy, or that are common to different applications. On the other hand, the various explicit `import` directives are meant to get a foreign executable code

and/or other symbol and use it locally. It is just natural to use both in a complex Falcon applications that may need application-aware components and then load utilities that are used locally in the context of a single module.

Dynamic module loading

Falcon makes possible to dynamically load modules by two means: the `include` function and the Reflexive Compiler class.

Explaining them both is beyond the scope of this survival guide; the `include` function is explained in the core module reference, and the Reflexive Compiler is a class exported by the `compiler` standard Feather module. They both allow to import dynamically module honoring or ignoring their exports, at loader's choice.

The `include` function can be provided with a dictionary of strings, whose values be filled with the global symbols with matching names coming from the loaded module.

The Reflexive compiler gives a greater degree of control on the loaded module, which is represented by a Falcon class and can be queried for symbols, inspected, executed, modified and so on. Also, the compiler is able to compile Falcon code on the fly from a string.

The directive statement

The behavior of the compiler can be configured through the `directive` statement, which alters one or some of the internal settings of the Falcon compiler.

The definition of the statement is the following:

```
directive <directive1> = <value1>, ..., <directiveN> = <valueN>
```

More than one directive statement may be specified in a file.

The directives specified in a source file affects only the given file. Compilation of other files is not subject to the directives specified in a given source, even if they are compiled to fulfill a load request in that source.

It is possible to specify values for directives that will affect all the compiled files from falcon command line interpreter, with the `-D` option. For example :

```
[user@host]$ falcon -D strict=on script.fal
```

This command would compile `script.fal` and any other related script setting the `strict` directive to `on`. However, files being loaded but having being already compiled differently won't be recompiled. To be sure to compile all the scripts with the selected directives, add the `-f` option so to force recompilation.

In the rest of the chapter, the directives currently available and their effect is described.

Lang directive

The `lang` directive declares the (human) language in which the module is mainly written. It is useful for internationalization, so that the translation table compiler knows which languages not to include in the final translation, and the module loader knows which translation tables are embedded

in the module and need not to be searched.

For more details about this feature, read the paragraph about program internationalization on page115.

Strict directive and def statement

The strict directive forces explicit declaration of variables through the `def` statement. This is useful to have the compiler to perform checks against possible symbol name misspelling errors.

Once the strict directive is set to 'on', every assignment to unknown symbols must be prefixed by `def`

The `def` statement can be followed by a list of assignments separated by commas. Because of this, multiple assignments to undefined variables cannot happen when strict is in control:

```
directive strict=on

// define some variable
def var1 = nil, var2 = nil

// forbidden: var4 is undefined
var4, var1, var2 = one, two, three

// allowed, var1 and var2 have already been def'ed.
var1, var2 = one, two
```

Because of the `def` statement grammar definition, array assignments must be explicit.

```
a = 1, 2, 3 // alternative to [1, 2, 3]

// but with strict active
directive strict=on

def a = 1, 2, 3 // error
def a = [1, 2, 3] // ok
```

In strict mode, variables must be redefined in their own context; for example :

```
def a = 1

function alpha()
   def a = "alpha value"
  //...
end

function beta()
   def a = "beta value"
   //...
end
```

To import a global variable in a local context, the global statement becomes mandatory:

```
def a = 1

function alpha()
    global a   // importing A
   a = "alpha value"
   //...
end
```

Version directive

The directive version permits assigning a version for a given module. The value will be available for the program loading the FAM module and for the script itself.

The value associated to the version directive must be a single number, and can contain major, minor and patch version numbers encoded in an hexadecimal number. For example , version 2.3.15 can be expressed as

```
directive version=0x2030F
```

Minor and patch version numbers are limited to 255, but major version number can be any value.

From Falcon scripts, the current module version can be accessed with the vmModuleVersionInfo() function, which returns an array of three values (major, minor, patch). If the version directive has not been set, it will return three zeros. For example:

```
directive version=0x010203

function  version()
   ver = vmModuleVersionInfo()
   > @"My program, version $(ver[0]).$(ver[1]).$(ver[2])"
end

version()        // --> My program, version 1.2.3
```

Advanced topics

The vast majority of falcon language is now covered. There are still a few of issues that are not covered by the tutorial-like part of this manual, and that are covered here just for reference.

Variable aliases and pass by reference

Falcon provides a powerful method to handle variables indirectly. Instead of having the symbol of a variable immediately available, the it is possible to create an alias to a variable. References can also be used to pass parameters to the functions, making the function able to alter their value. A variable alias is created with the `$` alias operator:

```
var = "original value"
var_alias = $var         // aliasing var to var_alias
var_alias = "new value"
printl( var )            // now will print "new value"

function change_param( param )
   param = "changed"
end

var = "original"
change_param( var )      // pass by value
printl( var )            // still "original"

change_param( $var )     // pass by alias
printl( var )            // now is "changed"
```

References are "sticky": they remain bound to the referencing variable until another reference is assigned to them. To remove the sticky reference from a variable, it is possible to assign a `$$` to it, meaning just "stop being a reference to something". After this operation is done, the referencing variable will be set to `nil`, and further assignments to it will be considered as normal assignments.

> *It is not possible to return values by reference. To avoid having "floating references" to objects that may be no longer valid, any return value pointing to a reference is turned into a copy of the referenced item. Note that this doesn't mean that deep objects as arrays, dictionaries and instances will be duplicated; only the item itself is duplicated.*

Coroutines

Coroutines are routines that run concurrently in the same Virtual Machine. The VM executes some instructions from one coroutine, then it swaps it out and goes on executing some instructions of another coroutine and so on until the first coroutine is called again. From a user standpoint, it seems that that all the routines are running at the same time.

Coroutines are not OS level threads. Calling a function that can cause the physical machine to block will suspend the execution of all the coroutines. Also, even if the target platform is provided with more than one CPU, all the coroutines will use the one on which the Virtual Machine is running.

The `launch` statement creates a new coroutine:

```
launch  function_name ( [function parameters] )
```

The coroutine is executed by calling the specified function with the required parameters; the execution continues in the same context where `launch` is called at the next lines, while the coroutine is beginning its processing.

Coroutines can launch other coroutines. Each one is independent from its parent; from a Virtual Machine point of view, the launcher and the new coroutine are identical in every respect. The VM will terminate the execution only when the last coroutine that has been launched completes its operation, or when a `exit()` function is explicitly called by any of the running coroutines. Of course, coroutines can call other functions, methods, code blocks, even recursively.

A coroutine terminates its execution when the function that was originally launched returns; also, it may terminate in any moment if it calls the `yieldOut()` function.

> *The functions named in this chapter are all provided by the core module. Although the core module is physically stored in the virtual machine library, it must be created and linked in the VM as any other module. Embedding applications may override it partially or completely.*

Synchronization

Synchronization is a very important aspect under any parallel environment; however, coroutine parallelism is just a "fake" parallelism, and this allows some simplification with respect to a full multi-threading model: while running, each coroutine is completely owning the Virtual Machine. Reads and writes to shared variables can be considered atomic. A coroutine needing more data can swap out and require another coroutine to be executed by calling the `yield()` function; if it thinks that it won't have anything useful to do for a certain time, it can call the `sleep( seconds )` function and it will be called only after the timeout has expired.

> *The `sleep( seconds )` function can also be called when the VM has not started any coroutine; this will make it to be idle for the required time. The parameter may be a floating point number if a pause shorter than a second is required.*

If a coroutine is preparing a complex set of data on which other coroutines may depend upon, it can be prevented from being swapped out when the job is not complete by calling the `}}{{beginCritical ()` function; this will force the VM to continue executing the coroutine without any interruption until it signals it is done by calling either `yield()`, `sleep( seconds )` or `endCritical ()` functions. In the latter case, the coroutine may continue its processing if its timeslice was not completely consumed.

Finally, a coroutine may put itself in wait for other parts of the process to be completed. This is done by synchronizing on a semaphore object, created from the `Semaphore` class.

```
synch = Semaphore( <initial value> )
```

The semaphore as an integer initial value (zero if not provided) which regulates its behavior. When the value is greater than zero, waiting on the semaphore by using the `wait()` method will allow the coroutine to continue its processing and will contextually reduce the semaphore value by 1. When the value is zero, the coroutine will be swapped out and won't be swapped in again until the semaphore value is incremented with the `post()`. In the following example, we use a semaphore to start a coroutine at a time:

```
function coro( id, syn )
   syn.wait()
   printl( "Coroutine ", id, " started" )
end

semaphore = Semaphore()   // will create a semaphore initially set to 0

launch coro( 1, semaphore )
launch coro( 2, semaphore )
launch coro( 3, semaphore )

for i in [0:3]
   sleep( 1 )
   semaphore.post()
end

printl( "Main program end" )
```

This script creates a semaphore object that is initially set to 0. When the first coroutine tries to wait on the semaphore, it gets blocked and swapped out. So happens to the other coroutines. Then, the main coroutine sleeps a bit and post on the semaphore. The expected output from this script is as follows:

```
Coroutine 1 started
Coroutine 2 started
Main program end
Coroutine 3 started
```

When some coroutines are waiting and a semaphore is posted, the coroutine that had been waiting for first is the first one to be re-enabled as soon as the current coroutine swaps out. To have a semaphore waiter to immediately punch in after a `post()`, `yield()` must be called after that. The `sleep()` function has actually the same effect, and it also tells the VM that the coroutine calling it shouldn't be reactivated again before a certain time; so after the last semaphore post, as the loop ends, the main coroutine is allowed to proceed before the third coroutine can punch in.

Program internationalization

Some of the strings contained in a Falcon program may be automatically translated into a target language of choice. Using the `i` character in front of a string, that gets marked as an

`international string` and gets readied to be exported to an XML dictionary. Translators (generally humans) may take this extracted XML file and fill translations for a certain language. An utility called `fallc.fal` (Falcon language compiler) will then convert one or more translated XML files into a single `ftr` (Falcon translation).

At link time, the modules will be able to load a translated table instead of their original string table, and will then show the translated strings with 0 overhead.

The `falcon` command line tool can generate a translation file for a source or a binary module (either an already compiled .fam or a native . `dll` /. `so` module) through the `-y` option. It is then possible to set the language for an application through the command line option `-l` (applicable to `falcon` or `falrun`). If the language is present in the .ftr files existing besides the loaded modules, it will be used, otherwise the operation will fail silently.

The languages must be indicated through the 5 characters ISO language names, as `en_US`, `en_GB`, `it_IT`, `ja_JP`, and so on.

A source file willing to be internationalized should declare the language in which is written through the directive `language`. If not declared, the starting language will be set to `C` (none).

For example, let's internationalize a small sample file:

```
directive lang="en_US"

> i"Hello world!"

// Let's add some variable...
var = int( random() * 100 )

// using string expansion operator in international strings
> @i"You are $(var)% lucky."

> "Test complete"    // this string will stay untouched.
```

Saving the file as `inat.fal` and running the command

```
$ falcon -y inat.fal
```

The file `inat.temp.ftt` is generated. It is possible to change the name of the output template translation file adding the -o option; for example :

```
$ falcon -o my_template_file -y inat.fal
```

will save the empty translation table into `my_template_file`.

The contents of the template translation table looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<translation module="inat" from="en_US" into="Your language code here">
<string id="2">
<original>Hello world!</original>
<translated></translated>
</string>
<string id="7">
<original>You are $(var)% lucky.</original>
<translated></translated>
</string>
</translation>
```

Supposing to translate the program into French and Italian, we'll copy this template into two files; their name is not relevant, but it may be convenient to save them as `<module_name>.<lang_code>.ftt`.

So, we'll write `inat.fr_FR.ftt` as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<translation module="inat" from="en_US" into="fr_FR">
<string id="2">
<original>Hello world!</original>
<translated>Bonjour a tout le monde!</translated>
</string>
<string id="7">
<original>You are $(var)% lucky.</original>
<translated>Vous avez $(var)% de chances.</translated>
</string>
</translation>
```

and `inat.it_IT.ftt` as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<translation module="inat" from="en_US" into="it_IT">
<string id="2">
<original>Hello world!</original>
<translated>Buongiorno, mondo!</translated>
</string>
<string id="7">
<original>You are $(var)% lucky.</original>
<translated>Sei fortunato al $(var)%.</translated>
</string>
</translation>
```

It is not necessary to translate all the strings; if the `translated` block is empty, the original string is used instead.

The final step is that to compile the translation tables into a single translation file for a module. This is done through fallc.fal, which works as follows:

```
$ fallc.fal inat.it_IT.ftt inat.fr_FR.ftt
```

The command will inform you about an `inat.ftr` file being created for the `inat` module.

> *Fallc.fal will also check for escaped variables contained in the original strings to be present in the translations; this will prevent mistakes as forgetting a variable or mixing up its name. To turn off this feature, use the `-c` command line switch.*

Now, using the `falcon -l` option, we can change the language of our program:

```
$ falcon -l fr_FR inat.fal
Bonjour a tout le monde!
Vous avez 84% de chances.
Test complete

$ falcon -l it_IT inat.fal
Buongiorno, mondo!
Sei fortunato al 84%.
Test complete
```

Merging newer versions of the string table

It is possible that the source code which has been internationalized is changed after being translated in various languages. In that case, it is necessary to merge the existing translations with the new template generated by `falcon -y`. The `-m <merge_table>` option of `fallc.fal` loads a template and fixes one or more translations, applying the new string ids, filling the missing new strings and removing unused old ones.

Suppose to change our program adding a new internationalized string at the beginning:

```
directive lang="en_US"

> i"Program begin..."
//... rest unchanged
```

The following command sequence will fix already translated tables and generate a new working ftr.

```
$ falcon -y inat.fal
$ fallc.fal -m inat.temp.ftt inat.it_IT.ftt inat.fr_FR.ftt
$ fallc.fal inat.it_IT.ftt inat.fr_FR.ftt
```

Examining the translation tables, you will notice the updated indexes and the new string to be translated.

It is possible to provide a specific different output for the result of the merge using the `-o` option, but in that case `fallc` will write only one translation on the designed output stream.

Variable parameter passing

Falcon supports variable parameter function calling. The compiler never checks for the number of parameters passed to a called function to count up to those that the function declares. In other words, the following code is perfectly acceptable:

```
function varcall( param1, param2 )
  printl( "First parameter: ", param1 )
  printl( "Second parameter: ", param2 )
end

varcall( "one" )
varcall( "one", "two", "three" )
```

In the first call, `param1` assumes the value of `one`; `param2` is set to `nil` by the VM. In the second call, the third parameter is simply ignored.

In general, all the unused parameters will be set to nil. However, the target function can know the number of parameters it has been actually called with and can retrieve their value respectively with the functions `paramCount` and `paramNumber`:

```
function varcall()
    for i = 1 to paramCount()
       print( i )
        switch ( i )
          case 1: print( "st" )
          case 2: print( "nd" )
          case 3: print( "rd" )
          default : print( "th" )
        end
           // note: paramNumber parameter count is zero based.
       printl( " parameter: ", paramNumber( i - 1 ) )
    end
   printl( "End of function.\n")
end

varcall( "one" )
varcall( "one", "two", "three" )
varcall( "one", "two", "three", "four", "five" )
```

So, the target function is able to configure itself given the parameters that have been given it. It's also possible to declare parameters in the function header, and then use the paramNumber and paramCount functions to access to other parameters:

```
function varcall( p1, p2 )
   printl( "Required params: ", p1, " and ", p2 )
   print( "Complete list: " )
   for i = 0 to paramCount() - 1 step 1
      print( paramNumber( i ) )
      if i != paramCount() - 1: print( ", " )
   end
   printl( "." )
   printl( "End of function.\n")
end

varcall( "one" )
varcall( "one", "two", "three" )
```

This example shows two features: first, it's possible to access via `paramNumber()` those parameters that have been declared in the function header. Second, the minimal number of parameter returned by `paramCount()` is the number of declared parameters (that's why the first call ends up printing two items). The declared parameters are considered **mandatory** and they are filled and provided to the target function even if the caller does not provide them. The idea is that you should use them only for those parameters that the caller really should provide, leaving any optional parameter for `paramNumber()` to take.

> *The functions named in this chapter are all provided by the core module. Although the core module is physically stored in the virtual machine library, it must be created and linked in the VM as any other module. Embedding applications may override it partially or completely.*

Accessing variable parameters by reference

The use of reference variables and by-reference parameter passing has been previously explained. The Core module provides two functions that allow a function to interact with parameters that have been passed by reference without knowing them in advance.

The function `paramIsRef ( <id> )` returns 1 if the nth parameter (starting from zero) has been passed by reference. The function `paramSet ( <id>, <value> )` is able to set the nth parameter, as if it was directly set by the script:

```
function varcall( param1 )
   param1 = "some value"
   paramSet( 0,"other value" )
   printl( param1 ) // will print "other value"
end
```

If the nth parameter has been passed by reference, then `paramSet()` will also change the value of the called item:

```
value = "original"
varcall( value )
printl( "In main code: ", value ) // still original
varcall( $value )
printl( "In main code again: ", value ) // now changed
```

The indirect operator

It is possible to access local, global and imported symbols by name through the unary indirect operator (#). The string can contain a variable followed by an arbitrary sequence of valid accessors; when the indirect operator is applied (either to the literal string or to a variable containing it), the

value of the named variable is returned; if the variable is not defined in the VM, or if the accessors are invalid, the VM will raise an access error.

Actually, the string expansion operator is implemented through the indirect operator; so many of the sequences that can be used in string expansion can also be used with the indirect operator.

For example, consider the following code:

```
var = 2

object test
   prop = [ "zero", "first", "second", "third" ]
end

value = # "test.prop[ var ]"
printl( "Value is now: ", value )
```

As for the string expansion operator, the indirect operator will translate accessors even recursively (i.e. decoding `var` as 2 and then accessing the element 2 in the property of the `test` object), but it will return the desired value.

The indirect operator only supports reading from variables, or accessing them. Function or method calling is not supported.

As the indirect operator is an unary operator, it is possible to apply it more than once in a row to do indirect indirections:

```
value = 1000
value_id = "value"
p_id = "value_id"
> ## p_id        // prints 1000
```

Meta compilation

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

For example :

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

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

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

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

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

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

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

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

inst_one = one()
inst_two = two()

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

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

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

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

Attributes

Attributes are arbitrary compile-time symbols to which a static value is associated. Modules, functions, classes and objects can be provided with attributes, that can tell something about their functionality or be used for various purposes.

```
// a program
stand_alone: true
author: "Giancarlo Niccolai"

function test()
   group: "coolies"
   author: "Rookie"

   // do something...
end

// show attributes of this module
inspect( attributes() )

// show attributes of the test function
inspect( test.attributes() )
```

Module attributes can be inspected by other modules through the Compiler feather module, or by the C++ API in embedding applications.