The Falcon Programming Language

Survival guide

Giancarlo Niccolai

— Release 0.8.14 —

The Falcon programming language – Survival Guide.

Table of contents

Welcome

Welcome to this tutorial for the Falcon programming language. As the philosophy of Falcon is exactly "do it easy, do it fast, do it right", we'll move immediately to see what Falcon is and how to use it. Our goal is to make everyone able to use Falcon in no more than a couple of hours, and to be able to master its basic construct in no more than two days.

The basics

Falcon is what is commonly named 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 allow 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 as 1.34e15, indicating 134 followed by 13 zeros. Falcon also supports hexadecimal and octal numbers in standard C notation (0x... 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; 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 for the program, but that can be used to write information, warnings or just notes in the text. Falcon supports C language single line and block comments. A single line comment begins with two slashes (like //) 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, like "/*...*/". 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, and it may be used to indicate to the shell that the Falcon interpreter 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 combination of elementary constants, variables or simpler expressions 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 some operators that works 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.

Mathematical operators supported are addition (+), subtraction (-), multiplication (*), division (/), power (**) and modulo (%). The power operator can take a fractional number as second operand; in this way it's also possible to perform arbitrary roots. For example, 100 ** 0.5 will give 10 as result. 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 provides also the C “++” and “--” operators. Both prefix and postfix increment are supported.

One particularly important expression in Falcon is the *function call*. Function calls works as in mathematics: they are composed of a symbol near two round brackets which can contain a list of *parameters*, each of which may be an expression, 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 the parameters that are provided with 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. The stream can be managed by the embedding application, redirected to files, 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 appear on the screen; if you are running Windows NT, 2000 or XP write "cmd". Press enter, and voilà. 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 matters, 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 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 is possible to modify the order by which lines are executed. This is done by some statements that go generally under the name of "control structures", and are divided in two main categories: flux (or conditional) control structures and loop control structures. We'll begin from the former.

Conditional structures allow the execution of part of the script if a condition is verified, or otherwise acknowledged to be **true**. To verify if a condition is true is the responsibility of "relational" operators, and by their bigger brothers "logical operators". Relational operators put two variables or values in a relation and the expression becomes **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 should look like:

```
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 is elegant 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

> "Thank you for having told me"
```

Try this program and enter some numbers. Try also to enter a letter; you will 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 relational and logical operators. Logical operators are responsible for combining more 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 those ones will be considered 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 may override the precedence using parenthesis, 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. In both cases, the variable must also be not greater than 8. Variables may hold also a truth value:

```
a = b > 0 or c > 0
/* some code here */
if a: printl( "A is true" )
```

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

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

```
a = 1
if (a += 1) == 2: > "a is now 2"
if a -= 2: > "a is now 0, so this check is 'false' and this line won't be printed"
```

So, 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 "`?:`") is directly borrowed from the C language; the value of the whole expression will be the expression right after the question mark if the 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"

> is_process_done ? "Done" : "Still incomplete"

if is_process_done ? first_test() : second_test()
     printl( "One of the two tests had been successful" )
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 every 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 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. In this case, 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 be used to check also against strings:

```
switch month
   case nil
      > "Undefined"

   case "Jan", "Feb", "Mar"
      > "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 printl !!!" )

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

end
```

The value 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 also 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 (":"); anyhow, each case can be placed on a single line this way.

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 have not been yet introduced: it may be a function, a class or an object instance.

Predefined types are the following:

- NilType
- IntegerType
- NumericType
- RangeType
- MemBufType

- FunctionType
- StringType
- ArrayType
- DictionaryType
- ObjectType
- ClassType
- MethodType
- ExtMethodType
- ClassMethodType
- LibFuncType
- OpaqueType

Predefined types get the priority over classes and instances cases, so a case 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 in 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 control flux reaches the "end" keyword, 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 program control flux will execute the first statement right after the end keyword. The **continue** statement does the opposite: it brings the control flux 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.

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 is 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 also somewhat confusing for the human reader. Sometimes you'll want it anyhow, but you must have a good reason for having a `while` loop to 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` statement is exactly like the `while` statement, except for the fact that it does not declare a validity condition. The loop is executed forever, until a statement inside the loop exits it. Other than the `break` statement, a loop may be interrupted by an error raising or a `return` statement; we'll see more about those in the rest of the guide.

Operationally, and even internally, the `loop` statement is equivalent to a while statement of which the condition is always true. The `loop` keyword is meant to explicitly declare to the reader (and to the Falcon interpreter) that the reason to terminate the loop is provided somewhere inside the block.

Rewriting the while true example loop:

```
count = 0
loop

   if count > 100
      break
   end

   // do something here
   count += 1
end
```

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

Loop statement accepts an arbitrary statement on the same line where it's declared. For example:

```
count = 0
loop count++
   > count
   if count == 10: break
end
```

This allows to put "visually" a more relevant statement in the loop line, without the need for a ";". This is equivalent to putting what follows the `loop` statement on the next line. Look carefully:

```
count = 0
loop count < 10   // won't do what you think!
   > count
   count ++
end
```

Will have the effect to just check if `count` is less than 10 forever!

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. put 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 is 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, 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 as that: ",
        (alpha_beta + gamma) * 15 - 2 +
           psi + fi)

printl( "This is shorter" )
```

As it is not possible to put any statement inside any parenthesis, you don't have to worry about the fact that you may accidentally forget to close them; 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 good to be able 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 )
```

WARNING: 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. 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:

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

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; the other two statements are executed anyhow, and that seems not to 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 incongruence in the text flux, but at least it will warn about it) this code is safer than the one above, that may look to do a thing and just do another one. Anyhow, 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 this unless you are sure that 1) you can't get confused and 2) statements look better on a single line.

Constant declarations

It is sometimes 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 grammar 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 it were a variable:

```
const loop_times = 3

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

Remember that `const` can declare only constants that will be visible in the module currently being compiled, and 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
```

Actually, numeric values are given to these constants are generated by adding 1 to the previous numeric value (rounded down); so, it is 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 is 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.

Basic data structures

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, it may be declared also without parenthesis:

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

But in this way, it won't be 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
```

Thiese two statements do the same thing, but the first is less confusing and more elegant.

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
/* Some code here */
a, b, c = array
```

This will do the same thing, but having the items packed in one variable makes 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 n.", i,": ", array[i] )
   i++
end
```

The function `len` will return the number of items in the array. Array items provide also a Basic Object Method called `len` which allows extracting the length of the array through the “dot” object access operator; the above line may be

rewritten as:

```
while i < array.len(): > "Element n.", 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 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 item 1.

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

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 cases, the last index is inclusive (also 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 it evaluates into 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:

```
list[0:2] = b       // removes items 0 and 1, and places b in them
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 end). A little 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 ) // there isn't any "alpha" in a now.
```

The `arrayFilter` functions 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 function declaration 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 (they may seem like function calls), strings (they may get merged, as we'll see in the next chapter), sub-arrays (they may be interpreted as index accessor of the previous token) and so on, but when programming in a functional context, where function calls and in-line expression evaluation are rare if not forbidden, this second notation may be more natural.

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

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

In this case, without the comma, 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. This is an example of the string functionalities:

```
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
```

Strings can span over multiple lines; opening a string on a line and not closing it on the same line will result in a newline being inserted at the end of each line. Whitespaces (i.e. indents) are also inserted in the string.

```
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 )
```

This is meant for you to be able to put long text blocks directly inside the program.

> This “literal” multi-line text block is to be considered “deprecated”. It may be removed in next releases.

However, you may find it more simple to use “concatenated” strings in a C-like fashion. Two strings separated only by whitespaces (or newlines) are concatenated by the compiler at compilation time. So the above is equivalent to:

```
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.\n"
printl( longString )
```

You will have a long string on the console.

Falcon strings supports 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.\n"
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 print also 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 to do very rudimentary things, as 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()
```

Finally, it is possible to forcefully separate nearby strings using the dot-quote operator. Two joining strings are normally merged into one, even if they are on different lines. In dot-square declared arrays, in which each element is not separated from the others by a comma, it is possible to create two distinct strings declaring the second one with a dot immediately followed by a quote (without spaces in between). Compare:

```
unique    = .["Hello"   " "   "world"]
different = .["Hello"  ." "  ."world"]
```

The array **unique** contains a single element, an “Hello world” string. The array **different** contains three strings.

International strings

Falcon strings can actually 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 = "これは国際のストリング"
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 kind 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 in 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 `string` and its Unicode ID (in decimal format).

In case you need to internationalize your program, you may check out the section Program Internationalization on page 114.

String polymorphism

To store and handle efficiently strings, Falcon 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, Arabian, Hebraic, Hindi and so on, two bytes are required. For fancy symbols as i.e. 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, character size requirement changed. The string has been copied into a wider space. Now, twelve characters were 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 inserting 0-255 character values in each memory slot, which is declared to be 1 byte long. Inserting a character requiring more space in them will polymorph them causing 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.

- Also, strings can be saved to files both by considering just their binary contents 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.

Literal Strings

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

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

The differences with respect to the double quote strings are:

- literal strings cannot span multiple lines
- literal strings are not auto-chained ('A' 'b' is not equivalent to 'Ab').
- literal strings do not support any escape sequence EXCEPT for backslash-quote (\') which inserts a single quote in the literal string.

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 have already a meaning.

String expansion operator

Many scripting languages have a means to “expand” strings with inline variables, and 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 a 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.

To disambiguate potential symbol conflicts, a parenthesis can be used after the “$” symbol expansion marker like this:

```
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 should be wrapped the output in.
- the 'r' letter forces alignment to the right.
- A dot followed by a plain number indicates the number of decimals a variable should be represented with.

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 $(id:4):$(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, growing from two to three digits, while the right padding and fixed decimal count was applied to the value.

Formats can be applied also 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
```

Whose 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 expansion 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 and it 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. Classical functions 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( h, 1 )
> "World uppercased: ", strUpper( w )
```

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, 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:

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

It is also possible to remove all the elements using the `dictClear` function. Other interesting functions are dictKeys and dictValues, which create a vector containing respectively all the keys and all the values in the dictionary:

```
a = [ 1=>'1', 2=>'2', "alpha"=>0, "beta"=>1 ]
>> "Keys: "; inspect( dictKeys( a ) )
>> "Values: "; inspect( dictValues( a ) )
```

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`, which return a special object 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.

The for/in loop treats strings as a sequence of characters. To access a character in its numeric Unicode representation, a normal for loop would be preferable, as the star accessor may be used to access directly numeric values without the need to create temporary strings containing just one character at a time. However, for/in loop is slightly faster than a basic for loop in the case that there is the need to scan the string retrieving text character values. The continue dropping statement will work as expected, removing the current character in the loop, and the dot assign operator will change the current character to the one indicated.

The following example breaks a string, removing stray numbers and changing spaces into dashes in the meanwhile.

```
string = "H8ere is a st0ri9ng"

for c in string
   forfirst: print( "Scanning string: " )
   if c >= "0" and c <= "9"
      continue dropping
   end

   if c == " "
      .= "-"
   end

   print( c )
   formiddle: print( "," )
   forlast: printl( "." )
end

printl( "The string is now: ", string )
```

This script will print the following results:

```
Scanning string: H,e,r,e, ,i,s, ,a, ,s,t,r,i,n,g.
```

```
The string is now: Here-is-a-string
```

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.

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 piece of code that may be reused again and again by providing them with different values, called *parameters*. More formally, functions are relational operators that relates 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 codominion). But in the meanwhile, they are left free by the math definition to have a lot of fun :-).

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 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 possible to return also 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 calling oneself function 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.

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 "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 is 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 content 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 also have 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, it is 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)
```

Lambda Expressions

Formally, a *lambda expression* is a parametric expression that can be used to perform repeated calculations. The expression stays the same, while the parameters gets assigned from outside at each iteration and the expression is re-evaluated each time.

The formal declaration of lambda expression is the following:

```
lambda p1, p2..., pn => expression
```

For example, the "sum" lambda expression, which simply sums its parameters:

```
sum = lambda a, b => a + b
```

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( (lambda a, b => a + b)(2,2) )
```

An interesting use of lambda expressions is that of defining a single-expression anonymous functions:

```
sum = lambda a, b => a + b
printl( sum(2,2) )
```

Notice that lambda expressions are closures, so:

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

However, the most important usage of lambda expressions is in functional programming, that will be discussed in the next chapter; many functional constructs require a parametric expression to be provided to them; a single function name is actually a parametric expression (that gets parametrized through call parameters) but lambda expressions are often more adequate to express the operation that should be performed. For example, take the map functional construct which evaluates in a new vector of values, each one being the result of a calculus over a corresponding value in a source vector. Suppose that the square of each source element is needed:

```
squares = map( lambda x => x * x, [1, 2, 3] )
```

The map constructs evaluates to a list of item multiplied by themselves; compare with:

```
function multiply_by_self( value )
   return value * value
end

// ... some code possibly here

squares = map( multiply_by_self, [1, 2, 3] )
```

Although equivalent, this second sample is much less readable, and is also less respectful of the way functional program slices are meant to be read.

> At the moment, lambda expressions are implemented as one-statement-only anonymous functions, but in the future they may be optimized as special call frameless code slices, and they may get even functionally simplified (pre-evaluated) when used in functional sequences. So, when appropriate use lambda expressions instead of functions and anonymous functions, and you'll get a free performance boost when this feature gets implemented.

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 under 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.

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 (see page 43).

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 available also 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.

Evaluation operator

Since version 0.8.12, Falcon provides also 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

all( .[
   .[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, [lambda x => x*x, lambda 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( lambda 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 used also 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
```

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 lambdas instead of fully declared functions as follows:

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

More functional loop

Another function performing functional loops is `times`. Times is an extremely flexible function which can perform ranged loops (as for/in). As in floop, oob(0) and oob(1) have special meanings, causing the loop to be broken or restarted.

For example:

```
i=0
.[times ,[2:11] $i .[                          // Notice the ',' to tell the range
  .[iff .[lambda x=> x % 2 == 1  $i] oob(1) ] // skipping impairs
  .[printl 'Counting even... ' $i]
]]()
```

The times function is relatively complex and can be used with several different patterns. The complete guide can be found in the function reference.

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 useless 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:

- Assign oob status; the extended unary operator `^+` and the `oob` function return an out of band flat copy of the item.
- Remove oob status; the operator `^-` and the `deoob` function return a flat copy of the item with the out of band status removed (in case it was active).
- Test for oob status; the operator `^?` and the function `isoob` return true if the item they receive is out of band, and false otherwise.
- Invert the oob status; the operator `^!` returns an out of band flat copy of the item on the right if it was originally not out of band, and returns a non-out of band version if the item was originally out of band. There isn't a standard function corresponding to this 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.

Also, 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 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 adequate 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 of functional sequences and of message programming, where the caller may not be prepared to deal with exceptions coming from the processor code, but at times it comes useful also 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.

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
```

Using late binding, functional sequences as times loops can be rewritten without using external symbols:

```
.[times ,[2:11] &i .[
  .[iff .[lambda x=> x % 2 == 1  &i] oob(1) ]
  .[printl 'Counting even... ' &i]
]]()
```

Self-referencing local values.

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

```
sequence = .[ &multiplier 3 ]
sequence.multiplier = .[ lambda 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 lambda value) 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 create a sequence that prints the content of the sequence to which it is attached:

```
vector = .[ 'a' 'b' 'c' 'd' ]
vector.display = .[dolist .[printl "Item... "] &self ]
```

```
vector.display()
```

As `&self` is resolved at evaluation as any other late binding, it is even possible to cache the function somewhere and apply it to different sequences:

```
printme = .[dolist .[printl "Item... "] &self ]
v1 = .[ 'a' 'b' ]
v2 = .[ 'c' 'd' ]
v1.display = printme; v2.display = printme

v1.display(); v2.display()
```

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 called also "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.

> Scripting languages do not tend to provide *scope protection* to the properties, that is, preventing the external world from modifing them if not using a set of methods called accessors. However, 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 work also 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.max_msg - 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 page 73) and check its value with a normal branch.

Classwide methods

It is 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.

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 the functions and methods all the methods declared in the base and child classes.

Base method overloading

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 declared in a subclass is called “overloading”.

Once overloaded, 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) having 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 be accessed also 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, overloading may happen also 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.

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 implementor must do to solve this situation is called “initialization on first use” idiom (this is actually C++ terminology, but works great also 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:

- If an object reads (or writes) a property in another object of the same module...
- ...create an *accessor* for that property that...
- ...initializes the property if it's **nil,** and then returns it...
- ...call that accessor from the init block of the owner object...
- ...and use that accessor instead of the property in the body of every object.

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.

Init on load idiom

A possibly simpler approach is to force initialization of all the objects in one spot. In case of objects referencing one another and being declared in the same module, each object may provide a common method, called for example "**configure**", which takes care of performing local object initialization. Then, each init block may call a common procedure (say... "**configureModule**"), that will take care of calling all the object's configuration routines the first time it's called.

To determine which objects needs configuration, an attribute (say... "**configure**") can be used. Attributes are given to objects before their property initialization and init block is called, so the objects will be in the correct attributed set by the time this mechanism is needed. A broadcast on the configure attribute will cause all the configure methods to be called.

```
// This attribute will mark objects in need for module-load time configuration
attributes: configure

// This function just broadcasts the configure attribute from a static block,
// which means, just once.
function configureModule()
   static
      broadcast(configure)
   end
end

object first
   list = nil

   init
     // configures
     configureModule()
   end

   // configurator; can be called only once
   function configure()
     self.list = [ "one ", "two ", "three"]
   end

   has configure
end

object second
   sum = nil
```

```
   init
      // Ensure module is configured
      configureModule()

      // Reading first.list directly
      for elem in first.list
          self.sum += elem
      end
   end

   function configure()
     self.sum = ""
   end

   has configure
end

printl( "second.sum: ", second.sum )
```

While a pure accessor approach is more flexible, this idiom is more elegant and efficient, but it can be used only when the dependencies between objects at startup are not circular. In case dependencies are not well defined or if they may change when the project evolves, using accessors that will create single properties if they are still not readied is the only feasible solution, but... notice that having dependencies at startup is already rare ad usually avoidable with different design choices. Circular dependencies at object initialization time is very probably something that may indicate a design flaw. Nevertheless, there may be cases in which they are both useful and coherent.

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 that those 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.

Also, blessing (that is, calling the `bless` function on the dictionary) is necessary because dictionaries are meant to hold even 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 can be accessed also 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 are not referencing 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()
```

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.

Attribute model

In the Falcon object model, an *attribute* is a generic, application wide binary characteristic that any instance may either *have* or *have not* at a given time. Having a certain characteristic may also mean “to be *like* something”. The eternal philosophical dispute between who thinks it's better to be and the ones that think that it's better to have is beyond our scope of analysis, but for the Falcon object model to have a determined attribute means also to be something related to it, and the other way around. To have freedom means to be free, to be ready means to have readiness.

> In version 0.8.12 (Condor), attributes can be applied only to class instances and singleton objects; they cannot be applied to blessed dictionaries and other prototype-OOP entities.

Having a characteristic automatically makes an object to belong to the set of objects that have the same characteristic. For example, suppose that you have a network program that has an unordered set of clients; some of them are being updated with data to be sent by the application network layer. Those clients are *ready* (or have *readiness*). The application network layer will want to access all the objects representing clients ready to receive data, and send the queued data to them. Instead of having to deal with all the objects representing a client of the application, the network layer may query Falcon for all the objects having readiness and handle just them.

So, a Falcon object (or in general any instance) *having* an attribute *is* also something related to that attribute and *belongs* to the set of all the items being like that. So, attributes describe a triple state of existence of a given Falcon instance: giving an attribute to an instance means that it *has* that, it *is* that and it *belongs* to that.

Attribute definition and grammar

Attributes are special symbols that can be defined only at global module level. They can be created using the following statement:

```
attributes
   attribute1
   attribute2
   ...
   attributeN
end
```

Or in the abbreviated form:

```
attributes: attribute1, attribute2, ..., attributeN
```

An attribute can be given to an object or an instance with the following syntax:

```
give attr1, attr2, ..., attrN to instance1, instance2, ... instanceN
```

To remove one or more attribute from an instance, the not keyword can be placed in front of the attributes to remove; attributes giving and removal can be performed in the same statement as the following example show:

```
attributes: ready, open, alive
give ready, not open, alive to instance
```

Giving an attribute already owned to, or removing a missing attribute from an instance has no effect.

Other than the give statement, attributes can be given to or removed from objects also through the `giveTo` and `removeFrom` functions. The above statement can be rewritten as:

```
giveTo( ready, instance)
removeFrom( open, instance )
giveTo( alive, instance )
```

The removeFromAll function takes a single attribute as a parameters and remove it from all the objects it has been assigned to up to the moment when the function is called.

To test whether an object has an attribute or not, the `has` and `hasnt` relational operators are provided. Their grammar is the following:

```
<instance> has/hasnt <attribute>
```

For example, the following code may be used to test for presence or absence of the “openness” attribute:

```
if instance has openness
   > "The instance is open"
elif instance hasnt life
   > "the instance is both not open and not alive"
end
```

A function called `testAttribute()` is also provided to access functionally the `has` operator functionality. The code above can be rewritten as:

```
if testAttribute( instance, openness )
   > "The instance is open"
elif not testAttribute( instance, life )
   > "the instance is both not open and not alive"
end
```

Both give and has statements can also work with attribute names rather than with variables containing the attribute itself. For example:

```
give "openness" to instance
if instance has "openness"
   > "The instance is open"
   //...
end
```

Functions working with attributes cannot use strings as attribute names like that, but it is possible to access the attribute having a given name via the `attributeByName()` function, which returns the attribute corresponding to the string passed as a parameter, if it exists.

Caching attributes

Attributes are symbols working exactly as any other Falcon symbol. They actually an attribute item that can be assigned to any variable, in case of need. Attributes can be returned from functions, stored in arrays, exported from modules and serialized as any other Falcon variable.

For example, the following code:

```
a = random()
if a > 0.5
   give open to instance
else
   give closed to instance
end
```

is equivalent to this code:

```
attrib = random() > 0.5 ? open : closed
give attrib to instance
```

and to this:

```
give random() > 0.5 ? open : closed to instance
```

It is even possible to assign an arbitrary value to a symbol previously containing an attribute, but it is strongly advised not to do it. Doing things as

```
closed = open
```

may deliver a serious blow to the mental stability of the casual reader of your code.

Class and object default attributes

At the end of class and object declarations, an extra declaration called "the *has* clause" can define the default attributes the instances of a class should have at creation, or in case of objects, the attributes that the object has on startup. In case of inheritances, the topmost class or the final object has all the attributes declared by the subclasses.

See the following example:

```
attributes: ready, open, alive

class base1
   // ... properties and methods
   has ready
end

class base2
   // ... properties and methods
   has open
end

object final from base1, base2
   // ... properties and methods
   has alive
end

if final has alive and final has open and final has ready
   > "Bingo!"
end
```

Our final object will have all the three attributes. The has clause may also contain not clauses to remove attributes that may be given by subclasses (or by classes previously declared as ancestors in case of multiple inheritance). In the following example, final has only the `open` attribute:

```
class base1
   has ready
end

class base2
   has open, not ready, alive
end

object final from base1, base2
   has not alive
end
```

The `not ready` clause in `base2` clears the `ready` set from `base1`, and then the `not alive` clause in `final` clears the alive set by `base2`. If `base1` and `base2` inheritance priority was inverted, the first `not ready` clause would have had no effect, and then the `base1 has` clause would have added the `ready` attribute.

Inspecting attributes in instances

The *Basic Object Method* called `attribs()` returns an array containing all the attributes that have been given to an

object. Every object has it, even if it is not visible in the `inspect()` report, or even if provides tested over `attribs` reports false. It may be useful to know exactly which attributes are currently given to an instance, for example to give them to a “brother” instance. See this example:

```
attributes: ready, open, alive

class base1
   has ready
end

class base2
   has open
end

object final from base1, base2
   has alive
end

// create an instance of the base class,
// but give it all the attributes owned by final
inst = base1()
for attrib in final.attribs()
   give attrib to inst
end

inspect( inst.attribs() )
```

The attributed set

The attributed set it the set of all the instances currently having a certain attribute. It is possible to perform three operations on the attributed set: traversing it, caching it and iterating over it.

Other than that, it is possible to broadcast a message to an attributed set, but we'll discuss about message broadcasting in the next chapter.

To traverse an attributed set, that is, to visit all the objects that are part of a set, it is possible to use a for/in loop whose collection is the attribute itself.

For example, we'll create 10 instances to which an attribute will be given randomly; we'll then visit all the objects that have received the attributes through a for/in loop.

```
class base( id )
   id = id
end

attributes: life

// create the instances
database  = []
for id in [ 0 :10 ]
   instance = base( id )
   if random() > 0.5: give life to instance
   database += instance
end

// let's see who are the lucky ones
for item in life
   > "in set: ", item.id
end
```

> As you may have noticed, we have stored the instances in a sort of “database”. Despite of this, being part of an attributed set does not mean that an object is fully available for the VM; the attribute set relations are what goes under the name of “weak reference” to the owned objects. As long as the VM thinks an object should exist, it also exists in the attributed set, but when an object ceases its existence because the script clears all the variables pointing to it, the VM may decide to remove that object in any moment. When this happens, the object is cleanly removed also from the attributed set. In short, if you want to have instances that stay in your program for a long while, they must be stored somewhere: giving them an attribute is not enough.

In the above for/in loop, it is possible to use the `continue dropping` statement to remove the attribute that is being traversed from the object currently being observed. For example, to remove `life` from all the items with an `id` smaller than 5, the following code may be used:

```
for item in life
   if item.id < 5: continue dropping
end
```

The attributed set may be also be cached in an array for later inspection, to rebuild it at a later moment, for intersection operations or for any other need. The `having()` function returns an array containing all the instances having a certain attribute. Continuing the previous example, it is possible to know which are the instances still having `alive` with the following code:

```
inspect( having( life ) )
```

Finally, it's possible to extract an *iterator* from the attribute to traverse the set or to modify it. We'll discuss about this topic in the chapter dedicated to iterators at page 94.

Message oriented programming

Message oriented programming is a programming paradigm which supports message exchanges between the entities operating in the application rather than explicit calls and direct relationships. The advantage of message programming with respect to other programming paradigms is being able to declare that “something just happened”, allowing other parts of the program to manage what happened when/if needed.

Message programming is a well known programming paradigm, used mainly in GUI programming; however, as only a very small set of features are usually needed to address GUI programming, message programming is usually implemented through function libraries covering the precise needs of a certain ranges of applications, and usually it is tightly bound with the GUI model that is provided with those libraries.

Falcon provides a generalized message oriented programming model, which is integrated with the other elements of the language and is applicable to pure logic problems and general application work flow.

The model was only introduced in version 0.8.6, so it is rather sparse in its current implementation. Further features are planned and will be integrated in upcoming versions, so to make it a *complete paradigm*. A complete paradigm is a programming paradigm application in which it is possible to build a complete, arbitrary program. Falcon procedural and object oriented paradigms are “complete”, meaning that whole applications can be written using just the constructs and the techniques which are provided by one of those models. Falcon functional programming is nearly complete, as it lacks a few constructs to be fully independent from the rest of the language, but it can go quite far towards the goal of being used to create purely functional programs.

In this version, message programming is still not complete, but it provides unique features and greatly enriches the set of available solutions at disposal of developers. Currently, message oriented programming is implemented through two main constructs called *message broadcasting* and *event dispatching*. Other than that, the *sender* object provides further support and can be used to expand the paradigm coverage.

Message broadcasting

An attributed set, that is, all the objects that have been given a certain attribute, can be an object of an arbitrary message broadcasting. The broadcast function takes a single attribute or a sequence of attributes and sends a message

The order in which the objects receive the message is random; there isn't any priority across a single attribute message. For this reason, the second form of `broadcast` function is provided. To implement priority processing, it is possible to broadcast a sequence of attributes. In that case, all the objects having the first attribute will receive the message, and will have a chance to stop further processing, before any item having the second attribute is broadcasted and so on.

Broadcasting an attribute means to inspect the objects in the attributed set and, if they provide a callable member having the same name of the attribute, to call it. Those members are named *broadcast handlers*.

Other than the attribute (or the list of attributes) to be broadcast, the `broadcast` function can receive an arbitrary number of parameters. Those extra parameters are *repeated*, that is, passed as-is to the handlers.

The handlers receive the notification in no particular order. When programming a message-oriented program, it is necessary to assume that the message handlers *may* be called concurrently, even if actually they get called in turn.

A handler can declare to have consumed a broadcast message by returning `true`. When this happens, the `broadcast` function ceases to send notification to other handlers and returns immediately.

This small example shows a typical broadcast process:

```
attributes: receiving

class Processor( id )
   id = id

   // this is the handler for the receiving broadcast
```

```
   function receiving( number )
      if number = self.id
         > "This signal is consumed by me, id=", number
         return true   // stop processing
      end
      > "It's not me... ", self.id
      return false     // signal we couldn't process
   end

   // declare we're willing to receive the message
   has receiving
end

// create 10 processors
processors = []
for c = 1 to 10
    processors += Processor( c )
end

// ask for a number
number = 0
while number <= 0 or number > 10
   >> "Enter a number between 1 and 10: "
   number = int( input() )
end

// tell everyone what happened
broadcast( receiving, number )
```

The ability to “consume” a message so easily should not be considered as a way to just “prevent” other receivers to process the same message, as the order in which the messages are sent is unknown. It is just a small optimization for when one of the handlers has taken care of what happened, and the rest of the program can easily ignore the message.

The best way for a potential receiver to declare its willingness to actually receive the broadcast is that of giving or removing the attribute to itself. This spares useless calls to handlers that won't actually handle the message anyway.

If an handler not willing to received notifications is still needed in the attributed set for other reasons, instead of having the attribute removed from the object, it is possible to temporarily move the handler routine in another property of the same object, and set a non callable value in that member. For example:

```
attributes: receiving

class Processor( id )
   id = id
   _storage = self.receiving

   function receiving( value )
      > "Processor ", self.id, " received message: ", value
      return false
   end

   function activate( mode )
      self.receiving = mode ? self._storage : nil
   end

   has receiving
end
```

As the functions are filled in instances before properties are handled, by the time the storage property is created `self.receiving` has been already assigned with the method in the class. That code actually saves a copy of the method in a property (and it is even possible to call the method created this way in `storage`).

If `activate` method is called with a `true` value, the method saved in `self.storage` is reassigned to

`self.receiving`, otherwise the handler is set to `nil`. In this way, the broadcasting process will silently skip the inactive instance, even if it's still in the attributed set of the broadcast attribute.

For example, continuing the above program:

```
processors = []
for c in [ 0 : 10 ]
   p = Processor( c )
   if random() > 0.5: p.activate( false )   // deactivate on random basis
   processors += p
end

broadcast( receiving, "Only someone" )

// now invite everyone and repeat broadcast
for instance in receiving: instance.activate( true )
broadcast( receiving, "Everyone!!" )
```

While the preferred method to activate and deactivate receiving of broadcast is that of giving or removing the attribute to the potential receivers, this second method adds flexibility to the model.

Broadcasting to priority queues

Broadcast receivers are notified in no particular order, but it is possible to broadcast the same message over a list of attributes. In that case, the next attribute in the list will be broadcast only after all the handlers receiving the previous attribute has received the message, and none of them has “accepted it” (that is, has returned `true`).

The next example creates three types of receivers and distributes them randomly in three priority queues.

```
attributes: first, second, third

class Processor( id )
   id = id

   function first( value )
       > "Processor ", self.id, " received FIRST PRIORITY message: ", value
      return false
   end

   function second( value )
       > "Processor ", self.id, " received SECOND PRIORITY message: ", value
      return false
   end

   function third( value )
       > "Processor ", self.id, " received THIRD PRIORITY message: ", value
      return false
   end
end

// create the processors
processors = []
for c in [ 1 : 10 ]
   p = Processor( c )
   switch int(random() * 3)
      case 0: give first to p
      case 1: give second to p
      case 2: give third to p
   end
   processors += p
end
```

```
prioqueue = [first, second, third]
broadcast( prioqueue, "The same message for all" )
```

When the broadcasting is performed, the processors distributed in the three attributed sets gets called following the priority queue order. If one of them returned true, it would have prevented all the processors of a lower priority order to receive the message.

Broadcast replies

It is often useful to return data to the broadcast issuer. If the method accepting the broadcast returns an out of band item, that item will be returned to the caller of the broadcast function (with the out of band attribute removed).

For example:

```
attributes: storage

object Storer
   _storedData = nil

   function storage( value )
      >> "(Storer had ", value, ") "
      old = self._storedData
      self._storedData = value
      return oob( old )
   end

   has storage
end

> "First call: ", broadcast( storage, "First" )
> "Second call: ", broadcast( storage, "Second" )
> "Third call: ", broadcast( storage, "nil" )
```

Event marshaling

The automated broadcasting system based on attributes is both powerful and simple, and using it alone to cover all the needs of message programming based applications is quite tempting. However, if taken to the limit, the attribute broadcast model would soon fill programs with meaningless attributes, used possibly just in one instance to receive a single message.

Attributes have an high expressive power: they dynamically declare what an object *has*, what it *is* and to what it *belongs* all at the same time. Sending a message to all the objects that are *all alike* one to each other is a procedure which has a clear and expressive meaning.

But if the program needs to have many differentiated messages that should reach potential different targets it is tempting to write new attributes that have a meaning only as “messages”. Overloading the attributes with this “*just message*” meaning may dirty the code to the point that it is impossible to distinguish the significance of attributes as dynamic declaration of characteristics being possessed by the instances. Maybe, the possibility that the two meanings of “messages” and “characteristic” get intermixed and confused is even worse.

To avoid this and to allow the attributes to stay just characteristics that an instance may dynamically have, and to allow the broadcast process to stay just the process of notifying a given situation to the objects that are *all alike* under a certain point of view, the concept of “event” has been introduced.

Events are arrays (or sequences). The first element of the array is a string called the *event name* which identifies the event. The other elements in the array are the *event parameters*. So, broadcasting an event is a matter of using this kind of *idiom*:

```
broadcast( anAttribute, [ "eventName", evparam1, ... evparamN] )
```

A special function set, called *marshaling functions*, automatically turns the event in a local call for the self object. The most basic marshal function is `marshalCB`, which should be given three parameters: the incoming event to marshal, an optional prefix for event handlers and a default return value to return when an handler for the event is not found. If an handler is found, `marshalCB` returns the same value returned by the handler, otherwise it returns its third parameter.

The prefix parameter has been provided as many developers are accustomed to having a certain nomenclature for event handlers; generally, an event handler for an event called “someEvent” is called “onSomeEvent” or “when_someEvent” or generally with a name which indicates both that the method is an event handler and what kind of event it handles.

The following is a minimal example of event marshaling:

```
attributes: ready

object generator
   // broadcast receiver
   function ready( event )
      // eventual pre-processing

      processingValue = marshalCB( event, "on'", nil )

      // eventual post processing, like
      > "Processing result was: ", processingValue

      // return; say we consumed the broadcast if the event was managed
      return processingValue != nil
   end

   // event handler(s)
   function onDataIncoming( data1, data2 )
      // process it...
      return "I have processed '" + data1 + "' and '" + data2 + "'."
   end

   // we're born ready...
   has ready
end

// our event is...
evt = [ "dataIncoming", "Some value", "other value" ]
// tell this fact to everyone that IS ready
broadcast( ready, evt )
```

Actually, this example is not so minimal; in fact, the broadcast receiver is able to mangle the incoming event before dispatching it to the actual handlers, and can do post-processing on the return value, taking a final decision about the broadcast being consumed or not. These are actually all the elements needed in complex event processing, and usually it is not needed to have all these elements in the broadcast receiver.

You'll notice the small single quote (') in the `marshalCB` call, terminating the parameter prefix. That small quote indicates that past the prefix, the first letter of the name of the handled event is upper-cased. In fact, the event name is `dataReady`, while our handler is called `onDataReady`. Without that quote in the prefix, for this sample to work, the handler should have been called “ondataReady”, or the event should have been called “DataReady”.

A final note about the `marshalCB` function regards the return value for missing handlers, that is, the third parameter. In some cases, the caller of `marshalCB` may wish to know if an handler has actually been called, but the handlers may legitimately return any value. In that case, to be able to tell if an handler has actually been called, it is possible either to give an out of band item as third parameter of `marshalCB` or not providing it at all. Without the third parameter, if `marshalCB` doesn't find a valid handler for the desired event it will raise a *property not found* error. The error may be intercepted by the caller to determine if the event has been processed or not.

In some cases, the event handler is considered “complete”. That is, it is supposed by design that it receives and handles all the events that are possibly broadcast to objects of its type (that is, having the attributes it should have at

runtime). In these cases, `marshalCB` should **not** be given a default return parameter. In this way, programming errors such as sending wrong events to the wrong handlers will be exposed, while they would be silently hidden if `marshalCB` were given a default return value.

> With the indirect call functions provided by Falcon, as `callMethod`, it would be practically possible to rewrite a marshal function in two or three lines of Falcon code. However, the marshaling functions are meant not to alter the *self* and *sender* special objects, and this is a relevant feature in message processing. In other words, if manually marshaled through Falcon indirect calls, or through searches in a dictionary of handlers and so on, the handler function would see the `sender` object as the object that marshaled the message, which is usually `self`, and not as the object that has originally broadcast the message.

The *just marshal* idiom

In the sample of the previous paragraph we have a complete handling and marshaling of the given event. This is the most general case, but in the most common case the broadcast receivers are just required to dispatch events to the handlers.

The function marshalCBX is meant to be directly assigned to broadcast receivers. The prefix and default return value are the first and second parameters, while the incoming message is the third one, so that a Sigma (a callable array) containing marshalCBX, the prefix and the default return value can be given in the property initialization, and a broadcast just calls it adding the event as third parameter.

The marshalCB sample in the previous paragraph may be approximated with the following code:

```
attributes: ready

object generator
   // broadcast receiver
   ready = [marshalCBX, "on'", false]

   // event handler(s)
   function onDataIncoming( data1, data2 )
      // process it...
      > "I have processed '" + data1 + "' and '" + data2 + "'."

      // signal process completion and broadcast consumption.
      return true
   end

   // we're born ready...
   has ready
end

broadcast( ready, ["dataIncoming", "Some value", "other value" ] )
```

Notice that the default return value is set to `false` to allow the broadcaster to find another suitable receiver in case the object does not handle the event. Also notice that the responsibility to declare broadcast consumption is passed down to the event handlers; if they return true the broadcast is terminated.

In `marshalCBX` the default return value is a mandatory parameter; to have an error raised in case of missing handler it is possible to use the `marshalCBR` function, that requires only the prefix parameter and raises a property access error in case the event is not handled by the calling object.

The *inheritance marshaling* idiom

It is possible to create a whole hierarchy of event handlers by simply declaring a base class for them.

For example:

```
attribute: listening

class Listener
   listening = [marshalCBX, "on'", false]
   has listening
end
```

Subclasses and derived objects are free to add any “on*” method to handle messages that will be broadcast on the “listening” attribute (that is, to the ones that are listening).

It is not necessary to give an initial `has` clause, and subclasses may define new attributes for which broadcasts will be received. However, with this base class it is possible to “deviate” all the broadcast traffic to a single attribute, and manage all the message programming of an application through events. Consider the following module:

```
attributes: listening

class Listener
   listening = [marshalCBX, "on'", false]

   function activate( on_or_off )
      if on_or_off
         give listening to self
      else
         give not listening to self
      end
   end

   has listening
end

function shout( event )
   if event.type() == ArrayType
      return broadcast( listening, event )
   else
      message = [event]
      for param in [1:paramCount()]
          arrayAdd( message, paramNumber(param) )
      end
      return broadcast( listening, message )
   end
end

export shout, Listener
```

Using this simple module, an application may create listeners receiving any event generated by the shout function on their `on<Event>()` methods, reserving the attributes to specify only attributed sets, or eventually using broadcast directly on other attributes distinguishing attributed set signals from generic event processing.

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. *Lots* of *interesting* data, for sure, but still just data.

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

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 it's 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 have also 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 virtualizing 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| lambda => self.name.len()*100 ],
...   [nil, nil],
...   [ "Smith", nil ],
```

```
...   [ "Sams", 2500 ] )
: Object
```

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

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

They can also be accessed directly through the columnData method;

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

Table operations

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

The choice method feeds all the rows in the table to an external function for evaluation; the row that gets the highest value is then selected and returned. For 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     lambda x=>x],
   .[  3000   4999     lambda x=>x*0.95],
   .[  5000   6999     lambda x=>x*0.93],
   .[  7000      0     lambda x=>x*0.9])

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

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

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

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

We can build 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 overloading in class hierarchies and so on is that the parameter definitions may be changed live, as the program runs. A new row may be inserted, a new column may be added, the discount conditions or the level limits may be altered, and still our offer functional sequence would reflect the up-to-date status of the condition table.

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

Table pages

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

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

For example, suppose 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'     lambda x => x * 2.3/100    lambda x => x * 8.4/100 ]
         .[ 'business'  lambda x => x * 0.8/100    lambda x => x * 4.2/100 ]
         .[ 'premium'   lambda x => x * 3.2/100    lambda x => x * 5.3/100 ]
         ])

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

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

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

// A bit of calculation.

// what should a premium account get for 1000$ on 2007?
AccTable.setYear(2007)
> "Asking for 1000$ 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 department 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, lambda x, y => -x.timesWon.compare( y.timesWon ) )
      for id in [0:bidders.len() ]
         >> @ "[$(id)] $(bidders[id].name) with $(bidders[id].timesWon) victor"
         > bidders[id].timesWon == 1 ? "y" : "ies"
      end
   end
end
```

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

```
randomSeed( seconds() )

bt = BidTable( 100, 6 )

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

bt.game()
```

The method `bidding` of the `Table` class does more or less what the bidding function we have created does in a table: it calls a method stored in a column, recording the one that returned the highest value and returning it. However, the loop in the bid method of this sample is not just selecting the row with the algorithm returning the highest value; it also changes the status of each row. 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 )
```

Also, it is now 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 be also 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()  // ok also 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 = dict.first()
lastIter = dict.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.

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 won’t 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
while age == 0
   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.message )
      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
```

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. Also, through this kind of semantic, 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:

- Using an internal translation table that may be set up by embedding application; in this case, requesting module loading will cause the Module Loader to map predefined functions into the runtime space of the target module, or to load application specific modules instead.
- Using a search path that can be set by the embedding application, or in case of the command line interpreter, that can be set up in the command line or passed as the `FALCON_LOAD_PATH` environment variable.
- Searches for a physical file name matching the logical name which is provided. The name must be written in double quotes, and path separators must be forward slashes (/).

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.

The master script can then be written as:

```
load slave

shared
shared = "Initialization value from master"
printl("MASTER - a shared variable: ", shared )
slave_check_shared()

printl( "Done." )
```

If you try this script removing the first line (where there's share alone), the changes to `shared` in one module won't be reflected in the other (`slave_check_shared` will print `NIL`).

My suggestion is to use explicit import only for those symbols that are likely to be assigned first than accessed in the main body of the script.

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 specificator 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 don'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 "`to`":

```
import func from module as myFunc
...
myFunc()  // actually, it is an alias for func in the given module
```

In this way, it is also 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:

1. The `load` directive loads indiscriminately all the exported symbols in the target module; `import/as` loads only the needed symbols, and loads them even if not explicitly exported.
2. Using the `load` directive on a module makes its exported symbols visible to the whole application being created by the Virtual Machine link process. In other words, the module may just need that function in the global namespace, but internally, while the loaded symbol(s) may clash with some symbol declared in another module. Using this explicit aliasing import prevents this from happening.

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.

In this moment, it is not possible to unload a loaded module. Actually, it is possible provided certain requirements and usage patterns; this limitation is scheduled to be removed in version 0.9.

The directive statement

The behavior of the compiler can be configured trough 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 page 114.

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 be used also 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.

The *sender* object

Other than the `self` object, which refers to the current object being handled by some method, Falcon keeps track of the `sender` object, which is the object that called a certain method. If a method was called from the main module or from a non-method routine, `sender` assumes the `nil` value, while if a method is called from within another method, the *self* of the caller is turned into the `sender` of the called. Look at this example:

```
object one
   function callable()
      if sender = two
         printl( "No, two doesn't have the right to call me." )
      else
         printl( "Ok, you may call me." )
      end
   end
end

object two
```

```
   function call_one()
      one.callable()
   end
end

// calling one from main
one.callable()

// calling one from two
two.call_one()
```

Knowing the object that is calling a method can be quite useful to build special agents that can acknowledge each other and cooperate in a tight way. Even if OOP programming suggests to encapsulate every needed knowledge in a single class, so that it can be self-sustained in every situation, knowing the `sender` object may be useful in small scripts or self-contained applications to rapidly build up a behavior model that involves more than one class (or more than one object). Possibly, the most interesting usage of the `sender` object is that of testing the interface that it provides (with `provides` or `in` relational operators), by being able in this way to dialog with callers in a smart way:

```
object called
   function callable()
      if sender provides toString
         printl( "I've been called by ", sender.toString() )
      end
   end
end
```

Coroutines

Coroutines are routines that run concurrently at the same time 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, lambdas, 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 be called also 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 also 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, `pram1` 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 is also possible to declare some parameter 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 change also 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
```

Function and method call intercept

Falcon provides a powerful system to intercept calls to functions or methods of any kind, so that it's possible to mask them and redirect every call in the program.

As function calls are actually requests to the VM to “call” a variable holding a “function object”, it is possible to mangle the value of function objects, and thus alter the call sequence in various ways. For example, you may easily create a function vector like this:

```
vector = [ printl, inspect ]

object inspected
    property = "content"
end

for func in vector: func( inspected )
```

The first call should print something like “<?>”, while the second call will print some meaningful information on the inspected object.

However, this mechanism can be used in much more interesting way. For example, we may substitute printl with something more personalized:

```
thing_to_prepend = "I want to say: "

function intercept( value )
   static
      oldprintl = printl
      return
   end

   oldprintl( thing_to_prepend, value  )
end

intercept() // initialization
printl = intercept

printl( "this and that" )
thing_to_prepend = "and also: "
printl( "that and this" )
```

The first call to intercept is meant to tell it what printl was originally. Then we assign intercept to printl, and from that point on, this VM will acknowledge calls to printl as if they were calls to intercept.

> Notice that `printl` is referenced before it's assigned; even if it's referenced in the intercept() function, that reference counts as an “import” for the `printl` symbol. Remember that it may be useful to explicitly import used symbols before assigning them.

However, it's easily understandable that this approach has a limitation: the intercept function can only receive a number of parameters known in advance.

Falcon provides two keywords that allows code to actually intercept calls to other functions or methods: the **pass** and **pass/in** statements.

The **pass** statement substitutes the current execution context with that of the symbol described after the keyword. In other words, the current function call disappears from the VM, its local variables being destroyed and its parameters being converted as they were the parameters for the given symbol. The substitution is so radical that, for example, a debug stack trace or an error report would not show the intercepting function.

While this may be a bit confusing, the solution is extremely efficient; the cost in VM steps is exactly the cost of the intercepting instructions plus one (the pass itself). Here's an example:

```
function passed()
   printl( "Passed called with ", paramCount(), " parameters." )
   pass printl
end
passed( "as", " ", "calling", " ", "printl" )
```

As this simple test demonstrates, `printl` punches in the VM, removing `passed`. As a side effect, the return value of the `passed` function is directly returned by the VM:

```
function passed()
   pass other
   printl( "This statement cannot be executed" )
   return "impossible value"
end

function other( parameter )
   return parameter
end

printl( "Expecting to receive 'a': ", passed( "a" ) )
```

As the execution of passed() is removed from the VM as the pass statement is executed, the return value of the function called thereon is directly passed to the caller. Notice, it's not possible in this way to intercept and manipulate the value returning from the called function.

The **pass/in** statement prepares the VM to give the control back to the interceptor as soon as the called symbol returns; the return value is stored in a given variable, and can then be inspected and changed. However, this has an additional cost: the VM cannot swap out the interceptor, and so it 's forced to re-push all the parameters; also the local variables of the interceptor cannot be removed during the call.

This is an example:

```
function interceptor()
   pass square in result
   if result = 0: result = 1
   return result
end

function square( parameter )
   return parameter * parameter
end

printl( "Square of 2: ", square( 2 ) )
printl( "Square of 2 (intercepted): ", interceptor( 2 ) )
printl( "Square of 0: ", square( 0 ) )
printl( "Square of 0 (intercepted): ", passed( 0 ) )
```

The `paramSet()` may be used to alter the parameters entering a function, but the count of the parameters can't be changed. It's not possible to add a parameter before or after the parameters that have been passed to the interceptor.

It's possible to apply this technique masking a certain symbol as we did before:

```
oldPrintl = printl  // this time we use a global
printl = interceptor

function interceptor()
   for i = 0 to paramCount() - 1
       if paramNumber( i ) = nil
         paramSet( i, "<was nil>" )
      end
```

```
   end

   pass oldPrintl
end

printl( "one ", nil , " two" )
```

The use of pass and pass/in statements, together with the parameter access functions provided by the core module and applying the assignment to external functions provides a complete model for function intercepting, excluding the ability to change the actual number of passed parameters. Some functions in Falcon allow indirectly calling symbols providing an arbitrary array of parameters, so that this limitation can be overcome too, but pass and pass/in provide a much more efficient solution where the parameter sequence for the intercepted function has not to been changed or synthesized on the fly.

A couple of notable examples may still be of interest. First of all, a “generic” interceptor object, that may be used to intercept any function:

```
class interceptor( func )
   icept = func

   init
      return self.intercept
   end

   function intercept()
      print( "intercepted.\n" ) // we'll use printl, so here we 'print'
      pass self.icept
   end
end

/* requires that we import... */ printl
printl = interceptor( printl )
printl( "Test" )
```

The interceptor class overrides the default behavior of the class initializer; it would normally return the instance of the newly created object, but `init` may return everything it desires. In this case, it's more suitable to return a method – the interceptor – that will then be assigned to the item we want to intercept. This is just a touch of elegance, as it was just possible to store the instance of the interceptor class somewhere, and then assign the method to `printl` normally.

After the initial assignment `printl` is a method; the object it refers to has a memory of the intercepted function, that is, the original `printl`. A complete implementation should throw an error in init if the symbol is one of the symbols used by the class itself. If `printl` were used instead of `print` in the `intercept` method, the VM would have faced an endless loop, calling the interceptor forever.

Notice also that we must explicitly import `printl`, as the assignment statement “declares” `printl` as a symbol local to the module before it requires it to be imported.

Similarly, we may intercept a method, like this:

```
//copy here the declaration of the interceptor class

object obj
   function a_method( param )
      printl ( "Printl from method: ", param )
   end
end

obj.a_method = interceptor( obj.a_method )
obj.a_method( "Some value" )
```

As this example demonstrates, the method can be intercepted as it were a normal call; moreover, the last example shows a method intercepting another method...

In the end, it's always possible to resume the old symbol value, removing the interception. Changing the main code of the former example with this:

```
old_value = obj.a_method
obj.a_method = interceptor( obj.a_method )
obj.a_method = old_value
obj.a_method( "Some value" )
```

the second call is not intercepted. The knowledge needed to restore the intercepted function may be stored in the interceptor object, in the intercepted one or in the main program.

> With regards to this last operation; normally, assigning a method to a method will only lead to storing a method item into the ex-method, turning it into a property. So, obj.a_method = old_value would cause obj.a_method to become a property containing the method obj.a_method. This isn't as worrying as it may seem, as the method is just a pointer to the original object and to a function; any change in obj would immediately reflected into obj.a_method, and the VM is fully prepared to read any kind of object when decoding a property, including a method, and even including a method referring the same object the property is coming from. That's exactly the way method intercepting can work.
>
> However, the VM recognizes methods being assigned to the original objects, and stores the original function back in them. At this time, it's not clear if the extra 2 to 4 controls needed to check for this are more heavy than just transforming the ex-method into a normal property holding, by chance, a method... especially considering that these checks are done on every store-to-property opcodes, while the case of a method reassigned to itself seems reasonably rare, and the extra cost of handling methods-in-properties rather than just methods in the load-from-object is near to 0 (or even less than 0, as the VM **has** to store a method item created from the function and the original object in the A register).
>
> This is a performance check that shall be performed when the optimized VM is in place, as any measurement now would be useless, and this notice is left into the documentation as a future memo.

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 be used also 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
\]
```

Index