?

?

?

Could be

MAD

an abbreviation for

“Mobile Applications Development”

?

In fact, we have

"Mobile Applications Development - first steps"

so we could say something like

"The MAD - first steps“

Sounds interesting?

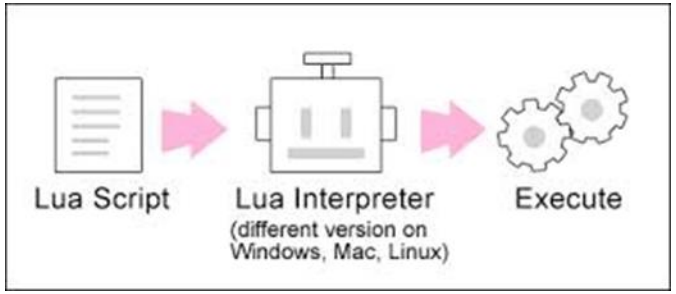
Mobile Applications Development

Course 3

a bit of

MAD episode 3 - a bit of Lua

What is Lua ?



- a programming language
- a scripting language
- a language with a good portability (can run on Unix, Windows, Windows CE, Symbian...)

Created in 1993 by Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes, members of the Computer Graphics Technology Group (Tecgraf) at the Pontifial Catholic University of Rio de Janeiro, Brazil.

MAD episode 3 - a bit of Lua

What is Lua ?

- A language written in ANSI C and ANSI C++ ie Lua is implemented as a library:
  - ⇒it's a case-sensitive language
  - ⇒hasn't a main function (as in C)
- A simple language-only one data structure-TABLES
- Has a small size: less than 200Kbytes ⇔ less than 20K lines of C code
- It is designed to support general procedural programming with data description facilities.
- It also offers good support for object-oriented programming and functional programming.

MAD episode 3 - a bit of Lua

How much does it cost ?

- Lua is a free software.

- It is available at Lua's official web site, www.lua.org.

Why should we learn Lua on our MAD course?

mainly because…

Corona SDK is based on it

Samples

```
--L1.lua:
print("Hello world")
```

```
--L2.lua:
function factorial(n)
  local x = 1
  for i = 2, n do
    x = x * i
  end
  return x
end

print(factorial(3))
```

Variables have only values, not types, so there aren't type definition, ie each variable carries its own type.

Lua is a dynamically typed language

- *Lua has eight basic types: nil, boolean, number, string, function, userdata, thread, and table.*

- **Nil** is the type of the value nil, whose main property is to be different from any other value; it usually represents the absence of a useful value.
- **Boolean** is the type of the values false and true.

Remark: nil and false make a condition false; any other value makes it true.

- **Number** represents real (double-precision floating-point) numbers. Lua hasn't an integer type !
- **String** represents immutable sequences of bytes. Strings can contain any 8-bit value, including embedded zeros ('\0').

- **functions**: can be called and manipulated both functions written in Lua or in C
- **userdata**: represent a pointer value to a block of memory
- **threads** represents independent threads of execution, used to implement coroutines.

Remark: Lua threads ≠ operating system's threads.

- **table,** a Lua type that implements associative arrays

Remarks:

- arrays that can be indexed not only with numbers, (any Lua value except nil and NaN (Not a Number, a numeric value that represent undefined or unrepresentable results, such as 0/0).
- tables can be heterogeneous (al types of values except nil).
- type function gives the type name of a given value
    - *print(type("Hello boys")) --> string*
    - *print(type(100.04)) --> number*
    - *print(type(print)) --> function*
    - *print(type(type)) --> function*
    - *print(type(true)) --> boolean*
    - *print(type(nil)) --> nil*

- the **table** type implements associative arrays.

ie arrays that can be indexed not only with numbers, but also with strings or any other value of Lua, except **nil.**

- tables are objects (dinamicaly objects), not values or variables
- tables have no fixed size; as many elements can be added dinamically.
- Lua programs manipulates only pointers (references) to them.
- In general, when a program hasn't a references to a table left, Lua's garbage collector will delete the table and free its allocated memory.
- A table can store values with different types of indices.

Example 1

```
1 myTable = {}      -- create a table and store its reference in ‘myTable‘
2 print(a) →00FA08B1 (memory address ⇔ pointer)
3 i = “ion“
4 myTable[i] = 9    -- new entry, with key=“ion" and value=9
5 myTable[20] = “romanul“          -- new entry, with key=20 and value=“romanul“
6 print(myTable[“ion"]) → 9
7 i = 20
8 print(myTable[i]) → “romanul“
9 myTable[“ion"] = myTable[“ion"] + 1          -- increments entry “ion“
10 print(myTable[“ion"]) → 10
11 print(i, myTable[i])        → ion romanul
12 yourTable = myTable     -- ‘yourTable’ refers to the same table as ‘myTable‘
13 print(i, yourTable[i])      → ion romanul
14 yourTable[“ion”]=99
15 print(myTable[“ion"]) → 99
```

Example 2

```
1 -- create an array/list (read elements from input console and store them into an array)
2 myTable={}
3 for i=1,10 do
4         myTable[i] = io.read()
5 end
3 for i=1,10 do
4         print(myTable[i])
5 end
```

- **Note: much more about tables can be found in "**Programming in Lua**"** by Roberto Ierusalimschy

Lua performs automatic memory management using *Garbage Collection* (like in Java)

ie

- User has to worry neither about allocating memory for new objects nor about freeing this memory when the objects are no longer needed.

- Lua manages memory automatically by running a garbage collector to collect all dead objects.

- Lua is a free-form language, ie spaces and comments are ignored,
- Identifiers can be any strings of letters, digits, and underscores, not beginning with a digit.
- The following keywords are reserved and cannot be used as identifiers in Lua:

and	*break*	*do*	*else*	*elseif*
end	*false*	*for*	*function*	*goto*
if	*in*	*local*	*nil*	*not*
or	*repeat*	*return*	*then*	
true	*until*	*while*		

- global
- local

Remarks:

- a variable name is assumed to be global unless it wasn't explicitly declared as a local;
- before the first assignment to a variable, its value is equal with nil value.

Statements in Lua contain:

- assignments,
- control structures (if, for, while)
- function calls
- variable declarations.

(like in C, Java, Pascal…)

- A **block** is a list of statements, which are executed sequentially. A *block* is the body of a control structure, the body of a function, or a chunk.

- The unit of execution of Lua is called a **chunk**. Syntactically, a chunk is simply a block. In fact, a chunck is an anonymous function

- A block can be explicitly delimited to produce a single statement:

do block end

Expressions

In Lua, expressions include constants, variables, operators (unary and binary) and function calls.

- **Arithmetic operators:**

- binary: +, -, *, /, ^ (exponentiation), % (modulo)
- unary `-' (negation).

All of them operate on real numbers.

So, in fact, a % b == a - floor(a/b)*b

- **Relational Operators**

  <, >, <=, >=, ==, ~= (not equal)

- **Logical Operators**

  and, **or**, and **not**.

print(14 and 9) → *14*
print(nil and 1) → *nil*
print(false and 1) → *false*
print(2 or 4) → *2*
print(false or 1) → *1*
print(not nil) → *true*
print(not false) → *true*
print(not 0) → *false*
print(not not nil) → *false*

Concatenation:

by .. (two dots):

print("ion " .. " romanul") → *ion romanul*

Notes:

-numbers are converted to strings:

print(0 .. 1) → *01*

-the concatenation operator creates a new string, without any modication to its operands:

a = "rom"
b=a.."anul"
print(b) → *romanul*
print(a) → *rom*

Operator precedence

^
not # - (unary)
*** / %**
+ -
..
< > <= >= ~= ==
and
or

Concatenation ('..') and exponentiation ('^') are right associative.

All other binary operators are left associative.

$$a = b + 7$$

Lua allows multiple assignments: a list of values(right side) is assigned to a list of variables (left side) in one step. Both lists have their elements separated by commas:

a,b,c = 1,2,3

i = 1

*i, a[i] = i+1, i*i/2*

a, b = b, a exchanges the values of *a* and *b*

a, b, c, d = b, c, d, a cyclically permutes the values of *a*, *b*, *c*, and *d*

Syntax:

```
if exp1 then
        block1
{elseif exp2 then
                block2}
        [else
                block3]
end
```

Example:

```
if op == "+" then
        r = a + b
elseif op == "-" then
        r = a - b
        elseif op == "*" then
                r = a*b
                elseif op == "/" then
                        r = a/b
                else
                        error("invalid operation")
end
```

Note: Lua has no switch statement!

Syntax:

- The **numeric for loop** repeats a block of code while a control variable runs through an arithmetic progression:

for var=exp1, exp2[, step] do
<block>
end

Remark: default step is 1, otherwise could be a value or a function/expresion

Example:

```
for i=1,10,2 do
        print(i)
end

for i=10,1,-1 do print(i) end
for i=1, f(x) do print(i) end
```

- The **generic for loop** works over functions, called iterators. On each iteration, the iterator function is called to produce a new value, stopping when this new value is nil:

for namelist in explist do
<block>
end

```
for i,n in ipairs(myArray) do
        print(n)
end
```

print all values of array ”myArray” using *ipairs* (a Lua iterator function used to traverse an array) ie *i* gets an index, while *n* gets the value associated with this index.

Syntax:

```
while condition do
        <block>
end
```

Example:

```
local i = 1
while a[i] do
        print(a[i])
        i = i + 1
end
```

Syntax:

```
repeat
        <block>
until condition
```

Example:

```
-- square root: using an algorithm and a  function
x = 81
sqrt = x/2
repeat
        sqrt = (sqrt + x/sqrt)/2
        local aprox = math.abs(sqrt^2 - x)
until aprox < x/10000
print (sqrt)
print (math.sqrt(x))
```

→

9.0000094155152

9

Why this difference? How can we improve the accuracy?

Syntax:

break

The break statement terminates the execution of a while, repeat, or for loop, skipping to the next statement after the loop

Example:

```
local i = 1
while myTable[i] do
if myTable[i] == x then break end
i = i + 1
end
```

return [explist]

- The return statement is used to return values from a function or a chunk (which is a function in disguise).
- Functions can return **more** than one value
- The return statement can only be written as the last statement of a block. Otherwise, a "*do return… end*" must be used.

Function's declaration:

```
function f()
....
 return a, b, c
end
```

Function's call:

```
x, y, z = f()
```

Functions

Syntax:

```
function name(param_list)
        <body>
        return…
end
```

Remarks:

-can *return multiple values*, by listing them after "**return**"

-may receive *a variable number of arguments*

-the parameter passing mechanism is *positional* (first argument gives the value to the first parameter, and so on)

-*lexical scoping:* a function enclosed in another function, has full access to local variables from the enclosing function (like global variables in C)

Example 1:

```
function minmax (a)             --an array as parameter
        local mini, maxi = 1, 1 -- index of the min max values
        local min, max = a[mini], a[maxi] -- min max values
        for i,val in ipairs(a) do
                if val < min then
                        mini = i; min = val
                end
                if val > m then
                        maxi = i; max = val
                end

        end
        return min, mini, max, maxi
end
print(minimum({1,3,4,12,4,0,33})) → 0  6  33  7
```

More about functions in: "Programming in Lua" by Roberto Ierusalimschy

Data Structures - arrays

In Lua the only data structures are **tables**. Structures like arrays, records, lists, queues, sets can be represented using tables. If we need :

1. **array**: indexing tables with integers

```
a = {} – name of new array
for i=1, 10 do
        a[i] = 0
end
```

- Any integer value is good for first/start index:

```
a = {}
for i=-10, 10 do
        a[i] = i
end
```

Data Structures- matrices

A matrix is an array of arrays

```
matrix = {} -- create the matrix (n x m)
n,m=10,5
for i=1,n do
        matrix[i] = {} -- create a new row in matrix
        for j=1,m do
                matrix[i][j] = 0
        end
end
```

Data Structures - Lists

Each node is represented by a table and links are simply table fields that contain references to other tables.

In fact, this particular list → is a …..?.....

Example: create a list by insert on front

```
list = nil
--insert first element at the begining of the list
list = {next = list, value = 0}
--insert other 3 elements:
for i=1,3 do
        list = {next = list, value = i}
end
local l = list
while l do
        print(l.value)
        l = l.next
end
```

→ *3 2 1 0*

Homework: Implement a queue (a double linked list) and “her” basic operations (push, pop).

Standard Libraries

Note: all libraries' functions can be seen in Lua reference manual.

Mathematical Library: math function (exp, log, log10, sin, cos, tan, asin, acos,....)

Example of use: math.abs (x)……

Table Library: functions to manipulate tables as arrays (insert, remove in lists, array's sort, string concatenation …)

Example of use: table.remove(myTable, 1) table.sort(myArray)……

String Library: manipulate strings

Example of use: string.len(s), string.upper(s), string.lower(s), string.find(s1,s2)…..

I/O Library: input/output files manipulation

Example of use: io.read(), io.input(file), io.output(file), io.open, io.write…….

Operating System Library: file manipulation, getting the current date/time, other facilities related to the OS.

Example of use: os.date, os.time, os.execute("mkdir"..dirname), …

Debug Library: ! Doesn't offer a debugger for Lua programs, offers only all the primitives needed for writing a debugger.

Example of use: debug.getinfo, ….

More can be said about the Lua language but it is enough for a course.

And much more can be read from www.lua.org

Ta-Ta for now!