A Linguagem de Programação Lua

Wilson Kazuo Mizutani
kazuo@uspgamedev.org

15 de abril de 2012

1. Introdução e breve histórico

Histórico

- Projeto da PUC-Rio (Tecgraf e LabLua)

Histórico

- Projeto da PUC-Rio (Tecgraf e LabLua)
- Site oficial: www.lua.org

Histórico

- Projeto da PUC-Rio (Tecgraf e LabLua)
- Site oficial: www.lua.org
- Últimas versões:

Histórico

- Projeto da PUC-Rio (Tecgraf e LabLua)
- Site oficial: www.lua.org
- Últimas versões:
  - 5.0 (obsoleto)

Histórico

- Projeto da PUC-Rio (Tecgraf e LabLua)
- Site oficial: www.lua.org
- Últimas versões:
  - 5.0 (obsoleto)
  - 5.1 (atual)

Histórico

- Projeto da PUC-Rio (Tecgraf e LabLua)
- Site oficial: www.lua.org
- Últimas versões:
  - 5.0 (obsoleto)
  - 5.1 (atual)
  - 5.2 (muito hipster ainda)

Principais características

Principais características

- Simplicidade

Principais características

- Simplicidade
- Eficiência

Principais características

- Simplicidade
- Eficiência
- Versatilidade

2. Sintaxe da linguagem Lua

Hello World!!!

Hello World!!!

helloworld.lua

```
1 -- My hello world Lua program:
2 print "Hello World!"
```

Hello World!!!

helloworld.lua

```
1 -- My hello world Lua program:
2 print "Hello World!"
```

Saída

```
Hello World!
```

Tipagem Dinâmica

Tipagem Dinâmica

- Comum em linguagens de script.

Tipagem Dinâmica

- Comum em linguagens de script.
- O tipo de uma variável é definido pelo valor atual dela.

Tipagem Dinâmica

- Comum em linguagens de script.
- O tipo de uma variável é definido pelo valor atual dela.
- Portanto, o tipo de uma variável pode mudar ao longo do programa.

O tipo e valor nil

O tipo e valor nil

nil.lua

```
1 -- x has a nil value.
2 x = nil
3 print(x)
4 print(type(x))
```

O tipo e valor nil

nil.lua

```
1 -- x has a nil value.
2 x = nil
3 print(x)
4 print(type(x))
```

Saída:

```
nil
nil
```

O tipo e valor nil

E agora?

nil.lua

```
1 -- x has a nil value.
2 -- x = nil
3 print(x)
4 print(type(x))
```

O tipo e valor nil

E agora?

nil.lua

```
1 -- x has a nil value.
2 -- x = nil
3 print(x)
4 print(type(x))
```

Saída:

```
nil
nil
```

O tipo e valor nil

E agora?

nil.lua

```
1 -- x has a nil value.
2 -- x = nil
3 print(x)
4 print(type(x))
```

Saída:

```
nil
nil
```

Importante! Variáveis não inicializadas (que "não existem") assumem automaticamente o valor de nil.

Valores booleanos

Valores booleanos

boolean.lua

```
1 -- x and y have boolean values.
2 x = false
3 y = true
4 print(type(x), x)
5 print(type(y), y)
```

Valores booleanos

boolean.lua

```
1 -- x and y have boolean values.
2 x = false
3 y = true
4 print(type(x), x)
5 print(type(y), y)
```

Saída:

```
boolean    false
boolean    true
```

Valores numéricos

Valores numéricos

number.lua

```
1 -- x and y have number values.
2 x = 42
3 y = .9e4 + 1
4 print(type(x), x)
5 print(type(y), y)
```

Valores numéricos

number.lua

```
1 -- x and y have number values.
2 x = 42
3 y = .9e4 + 1
4 print(type(x), x)
5 print(type(y), y)
```

Saída:

```
number   42
number   9001
```

Cadeias de caracteres (string)

Cadeias de caracteres (string)

string.lua

```
1 -- x and y have string values.
2 x = "hello"
3 y = [[
4 This is a multi-lined
5 text, which is very common
6 in scripting languages.]]
7 print(type(x), x)
8 print(type(y))
9 print(y)
```

Cadeias de caracteres (string)

string.lua

```
1 -- x and y have string values.
2 x = "hello"
3 y = [[
4 This is a multi-lined
5 text, which is very common
6 in scripting languages.]]
7 print(type(x), x)
8 print(type(y))
9 print(y)
```

Saída:

```
string  hello
string
This is a multi-lined
text, which is very common
in scripting languages.
```

Conversões automáticas entre numbers e strings

Conversões automáticas entre numbers e strings

conversion.lua

```
1 -- x has a number value and
2 -- y has a string value.
3 x = 13
4 y = "37"
5 -- what about z and w?
6 z = x + y
7 w = x .. y
8 print(type(z), z)
9 print(type(w), w)
```

Conversões automáticas entre numbers e strings

conversion.lua

```
1 -- x has a number value and
2 -- y has a string value.
3 x = 13
4 y = "37"
5 -- what about z and w?
6 z = x + y
7 w = x .. y
8 print(type(z), z)
9 print(type(w), w)
```

Saída:

```
number   50
string   1337
```

Funções

Funções

Função é um tipo nativo em Lua!

Funções

Função é um tipo nativo em Lua!

function.lua

```
1 -- f has a function value
2 f = function() end
3 print(type(f))
4 print(f)
```

Funções

Função é um tipo nativo em Lua!

function.lua

```
1 -- f has a function value
2 f = function() end
3 print(type(f))
4 print(f)
```

Saída:

```
function
function: 0x9d74b0
```

Tabelas

Tabelas

Tabelas são o principal tipo em Lua!

Tabelas

Tabelas são o principal tipo em Lua!

table.lua

```
1 -- t has a table value
2 t = {}
3 print(type(t))
4 print(t)
```

Tabelas

Tabelas são o principal tipo em Lua!

table.lua

```
1 -- t has a table value
2 t = {}
3 print(type(t))
4 print(t)
```

Saída:

```
table
table: 0x866780
```

Usando tabelas - Criando uma tabela com campos

Usando tabelas - Criando uma tabela com campos

tableusage1.lua

```
1 -- t receives a table initialized
2 -- with some fields.
3 t = {
4   10, 20, 30,
5   a = 40, b = 50,
6   [3.14159] = "PI"
7 }
8 -- Print t's contents.
9 table.foreach(t, print)
```

Usando tabelas - Criando uma tabela com campos

tableusage1.lua

```
1 -- t receives a table initialized
2 -- with some fields.
3 t = {
4   10, 20, 30,
5   a = 40, b = 50,
6   [3.14159] = "PI"
7 }
8 -- Print t's contents.
9 table.foreach(t, print)
```

Saída:

```
1       10
2       20
3       30
a       40
3.14159 PI
b       50
```

Usando tabelas - Inserindo campos

Usando tabelas - Inserindo campos

tableusage2.lua

```
1  -- t receives an empty table
2  t = {}
3  -- Inserting some fields.
4  t[1] = true
5  t[0.999] = "almost one"
6  t["a function!"] = function() end
7  -- Syntax sugar for string keys:
8  t.somefield = {}
9  -- Print t's contents.
10 table.foreach(t, print)
```

Usando tabelas - Inserindo campos

tableusage2.lua

```
1  -- t receives an empty table
2  t = {}
3  -- Inserting some fields.
4  t[1] = true
5  t[0.999] = "almost one"
6  t["a function!"] = function() end
7  -- Syntax sugar for string keys:
8  t.somefield = {}
9  -- Print t's contents.
10 table.foreach(t, print)
```

Saída:

```
1             true
a function!   function: ...
0.999         almost one
somefield     table: ...
```

Usando tabelas - Removendo campos

Usando tabelas - Removendo campos

tableusage3.lua

```
1 -- t receives a non-empty table.
2 t = { 10, 20, 30 }
3 -- Let's check it out.
4 table.foreach(t, print)
5 -- Removing middle field.
6 t[2] = nil
7 -- What about now?
8 table.foreach(t, print)
```

Usando tabelas - Removendo campos

tableusage3.lua

```
1 -- t receives a non-empty table.
2 t = { 10, 20, 30 }
3 -- Let's check it out.
4 table.foreach(t, print)
5 -- Removing middle field.
6 t[2] = nil
7 -- What about now?
8 table.foreach(t, print)
```

Saída 1:

```
1 10
2 20
3 30
```

Saída 2:

```
1 10
3 30
```

Usando tabelas - Removendo campos

tableusage3.lua

```
1 -- t receives a non-empty table.
2 t = { 10, 20, 30 }
3 -- Let's check it out.
4 table.foreach(t, print)
5 -- Removing middle field.
6 t[2] = nil
7 -- What about now?
8 table.foreach(t, print)
```

Saída 1:

```
1 10
2 20
3 30
```

Saída 2:

```
1 10
3 30
```

É o mesmo de antes: campos que valem 'nil' não existem!

Usando tabelas - Possibilidades e restrições

Usando tabelas - Possibilidades e restrições

- Praticamente QUALQUER COISA pode ser inserido em uma tabela de Lua, seja como chave ou como valor.

Usando tabelas - Possibilidades e restrições

- Praticamente QUALQUER COISA pode ser inserido em uma tabela de Lua, seja como chave ou como valor.
- Isso significa que tabelas podem ter outras tabelas e até mesmo funçoes como chaves e valores!

Usando tabelas - Possibilidades e restrições

- Praticamente QUALQUER COISA pode ser inserido em uma tabela de Lua, seja como chave ou como valor.
- Isso significa que tabelas podem ter outras tabelas e até mesmo funçoes como chaves e valores!
- A ÚNICA EXCEÇÃO é o 'nil'.

Usando tabelas - Possibilidades e restrições

- Praticamente QUALQUER COISA pode ser inserido em uma tabela de Lua, seja como chave ou como valor.
- Isso significa que tabelas podem ter outras tabelas e até mesmo funçoes como chaves e valores!
- A ÚNICA EXCEÇÃO é o 'nil'.
  - Chaves NUNCA podem ser 'nil' (isso causa um erro)

Usando tabelas - Possibilidades e restrições

- Praticamente QUALQUER COISA pode ser inserido em uma tabela de Lua, seja como chave ou como valor.
- Isso significa que tabelas podem ter outras tabelas e até mesmo funçoes como chaves e valores!
- A ÚNICA EXCEÇÃO é o 'nil'.
  - Chaves NUNCA podem ser 'nil' (isso causa um erro)
  - Valores 'nil' são removidos da tabela.

Threads

Threads

- Threads se parecem com funções, só que elas podem ser executadas sincronizadamente (i.e. paralelamente ao mesmo tempo)

Threads

- Threads se parecem com funções, só que elas podem ser executadas sincronizadamente (i.e. paralelamente ao mesmo tempo)
- Talvez não pareça, mas isso é bem complicado de usar!

Threads

- Threads se parecem com funções, só que elas podem ser executadas sincronizadamente (i.e. paralelamente ao mesmo tempo)
- Talvez não pareça, mas isso é bem complicado de usar!
- Por isso não abordarei threads nesse curso. Os curiosos podem fuçar o manual do Lua e explodirem seus computadores à vontade =P.

Userdata

Userdata

- Userdata representam objetos nativos que vieram de fora do Lua.

Userdata

- Userdata representam objetos nativos que vieram de fora do Lua.
- Portanto, não podem ser criados de dentro dele.

Userdata

- Userdata representam objetos nativos que vieram de fora do Lua.
- Portanto, não podem ser criados de dentro dele.
- E sua forma de uso é totalmente determinada por quem o criou.

Userdata

- Userdata representam objetos nativos que vieram de fora do Lua.
- Portanto, não podem ser criados de dentro dele.
- E sua forma de uso é totalmente determinada por quem o criou.
- Por exemplo, ele poderia ser um objeto associado a uma imagem, e ter métodos "draw(x,y)" ou "rotate(degrees)".

Escopo global

Escopo global

- Por padrão, as variáveis são todas globais, isso é, elas pertencem ao escopo global.

Escopo global

- Por padrão, as variáveis são todas globais, isso é, elas pertencem ao escopo global.
- O escopo global nada mais é do que uma tabela que sempre existe e à qual todos têm acesso naturalmente.

Escopo global

- Por padrão, as variáveis são todas globais, isso é, elas pertencem ao escopo global.
- O escopo global nada mais é do que uma tabela que sempre existe e à qual todos têm acesso naturalmente.
- É possível acessá-lo mais explicitamente através da variável global "_G".

Escopo global

- Por padrão, as variáveis são todas globais, isso é, elas pertencem ao escopo global.
- O escopo global nada mais é do que uma tabela que sempre existe e à qual todos têm acesso naturalmente.
- É possível acessá-lo mais explicitamente através da variável global "_G".
- Sim, o escopo global está inserido nele mesmo.

Escopo local

Escopo local

local.lua

```
1  -- x will not be a global variable.
2  local x = 0
3  y = 5
4  do
5    local x = 10
6    do
7      y = 20
8      print(x, y)
9    end
10 end
11 print(x, y)
```

Escopo local

local.lua

```
1  -- x will not be a global variable.
2  local x = 0
3  y = 5
4  do
5    local x = 10
6    do
7      y = 20
8      print(x, y)
9    end
10 end
11 print(x, y)
```

Saída 1:

```
10    20
```

Escopo local

local.lua

```
1  -- x will not be a global variable.
2  local x = 0
3  y = 5
4  do
5    local x = 10
6    do
7      y = 20
8      print(x, y)
9    end
10 end
11 print(x, y)
```

Saída 1:

```
10    20
```

Saída 2:

```
0     20
```

Escopo local

local.lua

```
1  -- x will not be a global variable.
2  local x = 0
3  y = 5
4  do
5    local x = 10
6    do
7      y = 20
8      print(x, y)
9    end
10 end
11 print(x, y)
```

Saída 1:

```
10    20
```

Saída 2:

```
0     20
```

Isso fará mais sentido daqui a pouco.

Atribuição

Atribuição

assignment.lua

```
1 -- Simple assignment.
2 x = 10
3 y = 20
4 -- Complex assignment
5 a, b, c = true, "sometext", {}
6 print(a, b, c)
7 -- Cross assignment. What happens?
8 x, y = y, x
9 print(x, y)
```

Atribuição

assignment.lua

```
1 -- Simple assignment.
2 x = 10
3 y = 20
4 -- Complex assignment
5 a, b, c = true, "sometext", {}
6 print(a, b, c)
7 -- Cross assignment. What happens?
8 x, y = y, x
9 print(x, y)
```

Saída 1:

```
true  sometext  table: ...
```

Atribuição

assignment.lua

```
1 -- Simple assignment.
2 x = 10
3 y = 20
4 -- Complex assignment
5 a, b, c = true, "sometext", {}
6 print(a, b, c)
7 -- Cross assignment. What happens?
8 x, y = y, x
9 print(x, y)
```

Saída 1:

```
true  sometext  table: ...
```

Saída 2:

```
20  10
```

Atribuição

assignment.lua

```
1 -- Simple assignment.
2 x = 10
3 y = 20
4 -- Complex assignment
5 a, b, c = true, "sometext", {}
6 print(a, b, c)
7 -- Cross assignment. What happens?
8 x, y = y, x
9 print(x, y)
```

Saída 1:

```
true  sometext  table: ...
```

Saída 2:

```
20  10
```

Variáveis em excesso ficam com 'nil'.
Valores em excesso são ignorados.

Estruturas de controle: if..then..else

Estruturas de controle: if..then..else

ifthenelse.lua

```
1 -- Condition statement.
2 if x < 10 then
3   -- do something
4 elseif x < 20 then
5   -- do something else
6 else
7   -- do yet another something
8 end
```

Estruturas de controle: if..then..else

ifthenelse.lua

```
1 -- Condition statement.
2 if x < 10 then
3   -- do something
4 elseif x < 20 then
5   -- do something else
6 else
7   -- do yet another something
8 end
```

vartest.lua

```
1 -- A variable evalues to true if
2 -- and only if its value is neither
3 -- nil nor false.
4 if x then
5   -- x is neither nil nor false
6 else
7   -- x is nil or false
8 end
```

Estruturas de controle: if..then..else

ifthenelse.lua

```
1 -- Condition statement.
2 if x < 10 then
3   -- do something
4 elseif x < 20 then
5   -- do something else
6 else
7   -- do yet another something
8 end
```

vartest.lua

```
1 -- A variable evalues to true if
2 -- and only if its value is neither
3 -- nil nor false.
4 if x then
5   -- x is neither nil nor false
6 else
7   -- x is nil or false
8 end
```

Zero não é avaliado para falso!

Estruturas de controle: while e repeat..until

Estruturas de controle: while e repeat..until

while.lua

```
1  -- Loops while the condition is
2  -- true.
3  i = 1
4  while i <= 10 do
5    -- something that happens
6    -- 10 times
7    i = i + 1
8  end
9  -- Loops forever... not.
10 while true do
11   -- how many times?
12   i = i - 1
13   break
14 end
```

Estruturas de controle: while e repeat..until

while.lua

```lua
-- Loops while the condition is
-- true.
i = 1
while i <= 10 do
  -- something that happens
  -- 10 times
  i = i + 1
end
-- Loops forever... not.
while true do
  -- how many times?
  i = i - 1
  break
end
```

repeatuntil.lua

```lua
-- Loops until the condition is
-- true.
i = 1
repeat
  -- something that happens
  -- 10 times
  i = i + 1
until i == 10
-- Loops foerever... neither.
repeat
  -- and now, how many times?
  i = i - 1
  if i < 5 then break end
until false
```

Estruturas de controle: for

Estruturas de controle: for

iteratedfor.lua

```
1 -- Loops from i = 1 to i = 10
2 for i = 1,10 do
3   -- Loops through 1,2,...,10
4 end
5 -- Loops from i = 1 to i = 10
6 -- jumping by 2 at a time.
7 for i = 1,10,2 do
8   -- Loops through 1,3,...,9
9 end
```

Estruturas de controle: for

iteratedfor.lua

```lua
-- Loops from i = 1 to i = 10
for i = 1,10 do
  -- Loops through 1,2,...,10
end
-- Loops from i = 1 to i = 10
-- jumping by 2 at a time.
for i = 1,10,2 do
  -- Loops through 1,3,...,9
end
```

foreach.lua

```lua
-- Loops for each key-value
-- pair in table t
for k,v in pairs(t) do
  print(k, v) -- for example
end
-- Loops for each index-value
-- pair in table t. Only 1..n
-- indexes are accessed.
for i,v in ipairs(t) do
  print(i,v) -- same example
end
-- Respectively equivalent to:
table.foreach(t, print)
table.foreachi(t, print)
```

Operadores

Operadores

arithmetic.lua

```
1 -- Simple math
2 x = (10 * (42 + 7)) / (5 * 7 + 192)
3 -- Also power and module
4 y = "6.022" * 10 ^ 23 -- problem?
5 z = 19238479127491 % 5
```

Operadores

arithmetic.lua

```
1 -- Simple math
2 x = (10 * (42 + 7)) / (5 * 7 + 192)
3 -- Also power and module
4 y = "6.022" * 10 ^ 23 -- problem?
5 z = 19238479127491 % 5
```

relational.lua

```
1 -- Relational operators always
2 -- return boolean values
3 less, notless = (x < y), (y >= x);
4 equal = (x == y)
5 different = (x ~= y)
```

Operadores

arithmetic.lua

```
1 -- Simple math
2 x = (10 * (42 + 7)) / (5 * 7 + 192)
3 -- Also power and module
4 y = "6.022" * 10 ^ 23 -- problem?
5 z = 19238479127491 % 5
```

relational.lua

```
1 -- Relational operators always
2 -- return boolean values
3 less, notless = (x < y), (y >= x);
4 equal = (x == y)
5 different = (x ~= y)
```

logical.lua

```
1 -- 'not' inverts boolean evaluation
2 nottrue = not true
3 -- If x evaluates true, returns y,
4 -- else returns x.
5 condition1 = x and y
6 -- If x evaluates true, return x,
7 -- else returns y.
8 condition2 = x or y
```

Operadores

arithmetic.lua

```
1 -- Simple math
2 x = (10 * (42 + 7)) / (5 * 7 + 192)
3 -- Also power and module
4 y = "6.022" * 10 ^ 23 -- problem?
5 z = 19238479127491 % 5
```

relational.lua

```
1 -- Relational operators always
2 -- return boolean values
3 less, notless = (x < y), (y >= x);
4 equal = (x == y)
5 different = (x ~= y)
```

logical.lua

```
1 -- 'not' inverts boolean evaluation
2 nottrue = not true
3 -- If x evaluates true, returns y,
4 -- else returns x.
5 condition1 = x and y
6 -- If x evaluates true, return x,
7 -- else returns y.
8 condition2 = x or y
```

concatenation.lua

```
1 -- String concatenation.
2 s = "It is over "..9000.."!"
```

Operadores

Operadores

tableconstructor.lua

```
1 -- We've seen a few times already.
2 t = {}
3 -- Also:
4 r = {
5   -- lots of stuff
6 }
```

Operadores

tableconstructor.lua

```
1 -- We've seen a few times already.
2 t = {}
3 -- Also:
4 r = {
5   -- lots of stuff
6 }
```

length.lua

```
1 -- We can get the indexed size
2 -- of tables and strings.
3 t = { 5, 5, 5, 5, 5 }
4 n = #t -- length is 5
5 m = #"yo there" -- length is 8
```

Definindo e usando funções

Definindo e usando funções

functiondef.lua

```
1 -- f takes x and returns x+10
2 f = function(x)
3   return x + 10
4 end
5 -- Another way to do it:
6 function ls(t)
7   table.foreach(t, print)
8   -- this one returns nothing
9 end
```

Definindo e usando funções

functiondef.lua

```
1 -- f takes x and returns x+10
2 f = function(x)
3   return x + 10
4 end
5 -- Another way to do it:
6 function ls(t)
7   table.foreach(t, print)
8   -- this one returns nothing
9 end
```

functionusage.lua

```
1 -- The result is 15
2 result = f(5)
3 -- One passing a single string
4 -- argument, parenthesis can be
5 -- omitted.
6 print "single string arg"
7 -- The same goes for single table
8 -- arguments.
9 ls { "first", "second", "third" }
```

Definindo e usando funções

Definindo e usando funções

Quantidade de parâmetros e retornos indefinida.

Definindo e usando funções

Quantidade de parâmetros e retornos indefinida.

functionvardef.lua

```
1  -- Functions may receive a
2  -- variable number of arguments
3  function f(x,y,z,...)
4    local a, b, c = ...
5    return a*x + b*y + c*z
6  end
7  -- And they may return many
8  -- values aswell.
9  function g()
10   return "hey","listen"
11 end
12 -- Or no value at all.
13 function h() end
```

Definindo e usando funções

Quantidade de parâmetros e retornos indefinida.

functionvardef.lua

```
1  -- Functions may receive a
2  -- variable number of arguments
3  function f(x,y,z,...)
4    local a, b, c = ...
5    return a*x + b*y + c*z
6  end
7  -- And they may return many
8  -- values aswell.
9  function g()
10   return "hey","listen"
11 end
12 -- Or no value at all.
13 function h() end
```

functionvarusage.lua

```
1 -- Using variable args.
2 f(1,2,3) -- error
3 f(1,2,3,4,5,6) -- ok
4 f() -- error, but possible
5 -- Getting multipe returns.
6 a,b = g() -- ok
7 c = g() -- "listen" is lost
8 d = h() -- guess what? nil.
```

Definindo e usando funções

Definindo e usando funções

Métodos!

Definindo e usando funções

Métodos!

method1.lua

```
1  -- A method is a function whithing
2  -- a table that takes the table
3  -- itself as its first argument:
4  t = { "a", "b", "c" }
5  function t.ls(self)
6    table.foreach(self, print)
7  end
8  -- But then we have to do this:
9  t.ls(t)
10 -- Which is annoying. Lua has
11 -- a syntax sugar for that:
12 t:ls()
```

Definindo e usando funções

Métodos!

method1.lua

```
1  -- A method is a function whithing
2  -- a table that takes the table
3  -- itself as its first argument:
4  t = { "a", "b", "c" }
5  function t.ls(self)
6    table.foreach(self, print)
7  end
8  -- But then we have to do this:
9  t.ls(t)
10 -- Which is annoying. Lua has
11 -- a syntax sugar for that:
12 t:ls()
```

method2.lua

```
1 -- The syntax sugar also works
2 -- for method definitions, in
3 -- which case the first argument
4 -- will be implicit, being
5 -- called 'self'
6 function t:size()
7   return table.getn(self)
8 end
```

Blocos

Blocos

- Um script lua apresenta uma hierarquia de blocos.

Blocos

- Um script lua apresenta uma hierarquia de blocos.
- Um bloco é uma sequência de comandos, como os exemplos que vimos até agora.

Blocos

- Um script lua apresenta uma hierarquia de blocos.
- Um bloco é uma sequência de comandos, como os exemplos que vimos até agora.
- Um bloco pode ter outros blocos dentro de si, através de:

Blocos

- Um script lua apresenta uma hierarquia de blocos.
- Um bloco é uma sequência de comandos, como os exemplos que vimos até agora.
- Um bloco pode ter outros blocos dentro de si, através de:
  - estruturas de controle (if, while, for, etc);

Blocos

- Um script lua apresenta uma hierarquia de blocos.
- Um bloco é uma sequência de comandos, como os exemplos que vimos até agora.
- Um bloco pode ter outros blocos dentro de si, através de:
  - estruturas de controle (if, while, for, etc);
  - definições de funções; ou

Blocos

- Um script lua apresenta uma hierarquia de blocos.
- Um bloco é uma sequência de comandos, como os exemplos que vimos até agora.
- Um bloco pode ter outros blocos dentro de si, através de:
  - estruturas de controle (if, while, for, etc);
  - definições de funções; ou
  - uso explícito de "do..end".

Visibilidade

Visibilidade

- Variáveis globais são visíveis a partir de qualquer bloco, contanto que não esteja obscurecida por alguma variável local visível de mesmo nome.

Visibilidade

- Variáveis globais são visíveis a partir de qualquer bloco, contanto que não esteja obscurecida por alguma variável local visível de mesmo nome.
- Variáveis locais são visíveis no bloco em que foram declaradas e nos blocos aninhados nele, contanto que sejam referenciadas após a linha em que foram definidas.

Visibilidade

- Variáveis globais são visíveis a partir de qualquer bloco, contanto que não esteja obscurecida por alguma variável local visível de mesmo nome.
- Variáveis locais são visíveis no bloco em que foram declaradas e nos blocos aninhados nele, contanto que sejam referenciadas após a linha em que foram definidas.
- Se um bloco tenta acessar uma variável que não está visível de jeito nenhum para ele, ele receberá um 'nil' na cara.

3. Bibliotécas básicas

Tópicos avançados que não aparecem nos slides

- Threads.
- Ambientes de funções.
- Metatabelas.
- API C.

Obrigado!

- Visitem www.uspgamedev.org
- Dúvidas: kazuo@uspgamedev.org
- Se seu interesse for desenvolvimento de games, vale MUITO a pena das uma conferida na engine LÖVE.
  - Site oficial: www.love2d.org
  - Jogo feito em LÖVE: http://stabyourself.net/mari0