LUA

JORDAN GONÇALVES PEREIRA
MATHEUS DE ABREU BOZZI
THIAGO DAMASCENO SILVA
VINÍCIUS DE OLIVEIRA RISSO

Apresentação da Linguagem

- Linguagem de Script de Multiparadigma;
- Interpretada e Compilada (gera um bytecode);
- Linguagem Brasileira, criada na Tecgraf (PUC-Rio) em 1993;
- Possui uma licença do MIT;
- Está na versão 5.3.5 (10 de julho de 2018);
- "Programação Orientada a Subterfúgios" ou, simplesmente, "Programação Orientada a Gambiarra";
- Bastante utilizada em Jogos;

Apresentação da Linguagem

- Diferente de **C** e outras linguagem, não precisamos utilizar “;” no final de cada linha de código;
- Podemos implementar **Lua** como uma **API C**;
- Para delimitar blocos, utilizamos palavras-chaves (reservadas), onde todos os blocos iniciam com uma delas (**for/do** e **if/then**, por exemplo) e terminam com a palavra **end**;
- Não é necessário definir o tipo da variável, todas são dinamicamente tipadas e são todas globais por *default*;

Apresentação da Linguagem

- Para comentar uma linha somente, basta utilizar dois **"-"** seguidos:

Exemplo de Código

```
a = 50
-- Esta linha está comentada
print(a)
```

Apresentação da Linguagem

- Para comentarmos um bloco de linhas, utilizamos
  –[[comentários]];
- Por praticidade, podemos utilizar da seguinte forma:

Exemplo de Código

```
--[[
<linhas_comentadas>
--]]
```

Exemplo de Código

```
---[[
<linhas_descomentadas>
--]]
```

Amarrações e Ambientes

Identificadores

- Palavras Reservadas:

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

Identificadores

- Lua é **case-sensitive**, ou seja:

  and ≠ And ≠ anD ≠ ...

- Identificadores podem ser qualquer combinação de letras, dígitos e underlines, desde que **não seja** iniciado por um dígito;
- É aconselhável a **NÃO** utilização de identificadores iniciados com underline seguidos de um ou mais caracteres maiúsculos;
- Estes, normalmente, são utilizados para casos especiais em Lua;

Amarrações e Ambientes

- Como mencionado anteriormente, todas as variáveis são globais, até mesmo em escopos locais;
- Para definir uma variável em um escopo local, basta utilizar o identificador **local**, neste caso, ela existe somente naquele bloco de execução;

Exemplo de Código

```
a = 1
for a < 10 do
    if a == 5 do
        x = 50          -- Variável Global
    end
    if a == 1 do
        local y = 1     -- Variável Local
    a = a + 1
end
```

Amarrações e Ambientes

- Se não atribuirmos um valor ao iniciar uma variável, ela possui **nil** como valor *default*, até que seja realizada a primeira atribuição;
- Lua permite atribuição múltipla:

Exemplo de Código
a, b, c = 10, “string”, 5.2

- Teremos **a** como **10**, **b** como **“string”** e **c** como **5.2**;
- Se não especificarmos a quantidade correta de valores a ser atribuídos, todas as variáveis excedentes “recebem” **nil**;
- Para deletar uma variável, basta atribuímos o **nil** para a mesma;

Amarrações e Ambientes

- Pode ocorrer sombreamento de variáveis:

Exemplo de Código

```
a = "global"
if true do
    local a = "local"
    print(a)
end
print(a)
```

- Saída:

Exemplo de Saída

```
local
global
```

Valores e Tipos de Dados

Valores e Tipos de Dados

- Como mencionado anteriormente, Lua é dinamicamente tipada, sendo assim, a variável recebe o tipo dependendo do valor atribuído a mesma:

Exemplo de Código

```
a = "global"
print(type(a))          → string
a = 10
print(type(a))          → number
print(math.type(a))     → integer
a = 10.5
print(type(a))          → number
print(math.type(a))     → float
```

Valores e Tipos de Dados

- **nil:**
  - Possui um valor único: **nil**;
  - Sua principal característica é ser diferente de todos os outros tipos, sendo uma espécie de *não-valorado*;
- **boolean:**
  - Possui dois valores: **true** e **false**;
  - **nil** e **false** fazem a condição ser falsa, enquanto qualquer outro valor a faz verdadeira;
  - Em particular, **0** e a string vazia são **true**;

Valores e Tipos de Dados

- **number:**
  - Até a versão 5.2, todos os **number** eram **double**;
  - A partir da versão 5.3, Lua começou a utilizar 2 tipos para representar os **number**: **integer** e **float**;
  - **integer** e **float** são "subtipos" de **number**;
  - Lua suporta constantes hexadecimais;
  - **Standard Lua** utiliza *64-bit integer numbers* e double-precision floating-point numbers (float);
  - **Small Lua** utiliza *32-bit integer numbers* e *single-precision floats*.

Valores e Tipos de Dados

- **string:**
  - Representam uma **imutável** sequência de bytes;
  - Lua é *8-bit clean*, ou seja, **strings** podem conter qualquer valor de *8-bit*;
  - Lua é *encoding-agnostic*, sendo assim, não faz suposições sobre o conteúdo da **string**;
  - **Strings** podem ser declaradas entre aspas simples ou aspas duplas;
  - **Strings** podem conter um único caractere ou um livro inteiro;

Valores e Tipos de Dados

- **function:**
  - Principal mecanismo de abstração de instruções e expressões em Lua;
  - Podem realizar uma tarefa especifica ou calcular e retornar valores;
  - São Valores de Primeira Classe com Escopo Léxico próprio;
  - Todas as funções em Lua são anônimas;
  - Será melhor tratado mais adiante em **Expressões e Comandos**;

Valores e Tipos de Dados

- **userdata:**
  - Permite que dados arbitrários de **C** sejam armazenado em variáveis de Lua;
  - Não possuem operações predefinidas em Lua, exceto de atribuição e teste de igualdade;
  - Utilizado para a **API C**;

Valores e Tipos de Dados

- **thread:**
  - Representação de *coroutines* em Lua;
  - *Coroutines* são similares a *Threads*;
  - Retomaremos sobre *Coroutines* mais adiante em **Concorrência**;

Valores e Tipos de Dados

- **table:**
  - Principal, na verdade único, mecanismo de estrutura de dados em Lua;
  - Podem ser utilizadas para representar Arrays, Conjuntos e várias outras estruturas de dados de forma simples, uniforme e eficiente;
  - Lua utiliza **tables** para representar pacotes e objetos;
  - Uma **table** em Lua é essencialmente um Array Associativo;
  - Para verificar a quantidade de elementos de uma **table** basta utilizar o operador **#**;

Valores e Tipos de Dados

- **table:**
  - Não são valores nem variáveis, são objetos;
  - Similares a Arrays em Java;
  - Podemos imaginar **Tables** como objetos alocados dinamicamente, onde programas manipulam apenas referências para eles;
  - **Tables** podem ter quaisquer tipos de valores dentro delas, inclusive outras **tables**, não sendo necessário que todos os elementos sejam do mesmo tipo;

Valores e Tipos de Dados

- **table:**

Exemplo de Intepretação	
> a = {}	-- Criação de uma Table Vazia
> k = “x”	
> a[k] = 10	-- Valor 10 colocado na key k=“x”
> a[20] = “great”	-- Valor “great” colocado na key 20
> a[“x”]	→ 10
> k = 20	
> a[k]	→ “great”
> a[“x”] = a[“x”] + 1	
> a[“x”]	→ 11

Valores e Tipos de Dados

- **table:**
  - Podemos acessar os valores postos em índices "*strings*" de uma maneira mais eficiente;

Exemplo de Intepretação	
> a = {}	**-- Criação de uma Table Vazia**
> a["x"] = 10	**-- Valor 10 colocado na key k="x"**
> a["x"]	**→ 10**
> a.x	**→ 10**
> a.y = "estou em y"	
> a["y"]	**→ "estou em y"**

  - **OBS:** a[x] ≠ a["x"]. Em a[x], x é uma variável; a["x"], x é uma **string** que representa um índice na **table**;

Valores e Tipos de Dados

- **table:**
  - Construtores:

Exemplo de Código

```
days = {“Domingo”, “Segunda”, “Terça”, “Quarta”, “Quinta”,
“Sexta”, “Sabádo”}

values = {x = 10, y = 20}
```

Valores e Tipos de Dados

- **table:**
  - **metatable**: *table* qualquer que permite a modificação do comportamento de outras tabelas em operações especiais;
  - Utiliza **metamétodos** para isso;
  - **setmetatable (table, metatable)** – associa uma *metatable* à uma *table*;
  - **getmetatable (table)** – informa qual *metatable* está associada a *table* em questão;

Valores e Tipos de Dados

- **table (metatable):**
  - Alguns **metamétodos**:

Metamétodo	Funcionalidade
__index	Muda a indexação table[key]
__newindex	Muda a atribuição de indexação table[key] = value
__call	Muda o comportamento de um chamado de table
__eq	Muda a operação de igualdade (==)
__tostring	Muda o comportamento da função **print**
__add	Muda a operação de soma (+)

Coleta de Lixo

Coleta de Lixo

- **Mark-and-Sweep Garbage Collector (V 5.0):**
  - Até a versão 5.0, Lua utilizava um coletor simples de Marcar e Varrer;
  - Este tipo de coletor também é chamado de "stop-the-world" collector. Isso se dá pelo fato de que, de tempos em tempos, o programa para de rodar para que o ciclo de coleta seja feito;
  - O ciclo compreende quatro fases: *mark*, *cleaning*, *sweep* e *finalization*;

Coleta de Lixo

- **Mark-and-Sweep Garbage Collector (V 5.0):**
  - **Mark:** Marca como ativo todos os objetos que consegue acessar;
  - **Cleaning:** Realiza a limpeza de *finalizers* e *weak tables*;

    O GC percorre todos os objetos marcados para finalização procurando por objetos não marcados, estes são ativados e colocados em uma lista separada para serem finalizados no final;
  - **Sweep:** Depois percorre todos os objetos, se um objeto não estiver marcado como ativo, o GC faz a coleta, caso contrário, o GC limpa sua marca, o preparando para o próximo ciclo;
  - **Finalization:** O GC faz a chamada dos finalizadores dos objetos que foram separados para limpeza;

Coleta de Lixo

- **Mark-and-Sweep Incremental Garbage Collector (V 5.1):**
  - Executa as mesmas etapas do anterior, mas não precisa "stop-the-world" enquanto é executado;
  - Ele é executado intercalando com o interpretador;
  - Sempre que o interpretador aloca uma certa quantidade de memória, o coletor executa uma pequena etapa;

Coleta de Lixo

- **Mark-and-Sweep Incremental with Emergency Collection Garbage Collector (V 5.2):**
  - Executa da mesma forma que o anterior, mas se caso uma alocação de memória falhe, é forçada um ciclo completo do GC e tenta realizar novamente a alocação;
  - Foi implementado, também, um **GC Geracional**, onde você podia fazer a troca de **Incremental** para **Geracional**;
  - Esta opção, Geracional, foi **retirada** na versão 5.3, pois era um modelo experimental para a versão 5.2;

Serialização

Serialização

- Lua escreve os dados como programas em Lua, que quando executados, reconstroem os dados;
- Lua possui várias biblioteca para utilizar o **JSON**, como *json*, *cjson*, *dkjson*, entre outras;
- Possui Serialização e Deserialização de *Tables* por meio dos métodos de *Table Serialization*;

Expressões e Comandos

Expressões e Comandos

- **Operadores Aritméticos:**

+	Soma
*	Multiplicação
//	Floor Division (Inteira)
^	Exponenciação

-	Subtração
/	Divisão
%	Módulo (Resto da Divisão)

- **Operadores Relacionais:**

<	Menor que
<=	Menor ou Igual a
==	Igual a

>	Maior que
>=	Maior ou Igual a
~=	Diferente de

Expressões e Comandos

- **Operadores Bitwise:**

&	Bitwise AND
|	Bitwise OR
~	Bitwise EXCLUSIVE-OR
>>	Shift-Right
<<	Shift-Left
~	Bitwise NOT (unário)

Expressões e Comandos

- **Operadores Lógicos:**

and	AND Lógico
or	OR Lógico
not	Negação Lógica

- **Operadores Unários:**

-	Inversor de Sinal
#	Operador Tamanho
~	Bitwise NOT (unário)
not	Negação Lógica

- **Operador de Concatenação:**

..	Concatena duas Strings

Expressões e Comandos

- **Ordem de Precedência:**

^					
-	#	~	not		
*	/	//	%		
+	-				
..					
<<	>>				
&					
~					
|					
<	>	<=	>=	~=	==
and					
or					

Operadores Unários

Bitwise exclusive OR

Expressões e Comandos

- **Operadores:**
  - Todos os operadores são *Left Associative*, ou seja, são operados da esquerda para direita;
  - Exponenciação e Concatenação são os únicos *Right Associative*, sendo assim, operados da direita para a esquerda;

Expressões e Comandos

- **and:** resultado é o **primeiro operando** se este é **falso**, **caso contrário** é o **segundo**;
- **or:** resultado é o **primeiro operando** se ele é **não falso**, **caso contrário** é o **segundo**;
- Os operadores **and** e **or** suportam curto circuito
- Embora não haja operador ternário como em **C**, ele pode ser facilmente implementado da forma:

Exemplo de Código	
a **and** b **or** c	**-- Equivalente em C = a ? b : c;**

Expressões e Comandos

- **if - then / elseif / else:**
  - Para utilizarmos este condicional, usamos da seguinte forma:

Exemplo de Aplicação

```
if exp1 then
        <alguma instrução>
elseif exp2 then
        <outra instrução>
else
        <outra instrução>
end
```

  - Funciona da mesma forma que em **C**;

Expressões e Comandos

- **while:**
  - Loop condicional pré-avaliado, ou seja, verifica a **exp** antes de executar as instruções;
  - Podemos utilizar da seguinte forma:

Exemplo de Aplicação

```
while exp do
        <alguma instrução>
end
```

  - Funciona da mesma forma que em **C**, repetindo o Loop enquanto a **exp** for verdadeira;

Expressões e Comandos

- **repeat - until:**
  - Loop condicional pós-avaliado, ou seja, verifica a **exp** após executar ao menos uma vez as instruções;
  - Podemos utilizar da seguinte forma:

Exemplo de Aplicação

```
repeat
        <alguma instrução>
until exp
```

  - Funciona da mesma forma que o **do – while** em **C**, repetindo o Loop enquanto a **exp** for verdadeira;

Expressões e Comandos

- **for numérico:**
  - Este **for** segue a seguinte sintaxe:

Exemplo de Aplicação

```
for var = exp1, exp2, exp3 do
        <alguma instrução>
end
```

  - Este **loop** irá executar **<alguma instrução>** enquanto o valor de **var** for de **exp1** até **exp2** usando **exp3** como incremento para **var**;
  - **exp3** é opcional e quando não é mencionado, tem valor **1** por padrão;

Expressões e Comandos

- **for numérico:**

Exemplo de Aplicação

```
for var = exp1, exp2, exp3 do
        <alguma instrução>
end
```

  - Se não quisermos um limite, podemos utilizar **math.huge**;
  - **var** existe somente durante o **loop**, sendo uma variável local;
  - Para manter o valor de **var** é necessário que uma outra variável criada fora do **loop**, por exemplo, receba seu valor durante o **loop**;

Expressões e Comandos

- **for genérico:**

Exemplo de Aplicação

```
for var-list in exp-list do
        <alguma instrução>
end
```

  - **var-list** é uma lista com um ou mais nomes de variáveis separados por virgulas;
  - **exp-list** é uma lista de uma ou mais expressões, também separadas por virgulas;

Expressões e Comandos

- **for genérico:**

Exemplo de Aplicação

```
for var-list in exp-list do
        <alguma instrução>
end
```

  - Usualmente, **exp-list** possui somente uma expressão que é uma chamada para um *iterator*;

Exemplo de Código

```
for k, v in pairs(t) do
        print(k, v)
end
```

Expressões e Comandos

- **break** e **return:**
  - Permitem que haja um "jump" para fora do bloco de execução;
  - **break** é utilizado para finalizar um loop (**for, while, repeat**), não podendo ser utilizado fora de um **loop**;
  - **return** "retorna" o resultado de uma função ou simplesmente finaliza a mesma;
  - Há um **return** implícito no final de cada função, mas estes não retornam a última linha válida como em outras linguagens;

Expressões e Comandos

- **goto:**
  - Adicionado na versão 5.2
  - Realiza um "jump" para um label correspondente;
  - O label é representado por um "identificador" entre dois pontos duplos.
    - Ex: **::label::**

Expressões e Comandos

- **function:**
  - Como foi dito anteriormente, **function** é a forma de abstrair instruções e expressões em Lua;

Exemplo de Código

```
function add (a)
        local sum = 0
        for i = 1, #a do
                sum = sum + a[i]
        end
        return sum
end
```

Expressões e Comandos

- **function:**
  - Funções possuem um nome, uma lista de parâmetros e um corpo, que possui a lista de instruções;
  - Os parâmetros funcionam exatamente como variáveis locais inicializadas com o valor dos argumentos passados na chamada;
  - Lua ajusta o número de argumentos passados para o número de parâmetros especificado, descartando os argumentos extras e aplicando **nil** para os parâmetros extras;

Expressões e Comandos

- **function:**

Exemplo de Código

```
function funparam (a, b)
    print (a, b)
end
```

Exemplo de Saída

```
funparam()          → nil    nil
funparam(3)         → 3      nil
funparam(3, 4)      → 3      4
funparam(3, 4 , 5)  → 3      4
```

Expressões e Comandos

- **function:**
  - Embora isto possa acarretar em problemas de programação, podemos utilizar isto para implementar argumentos *default*;

Exemplo de Código

```
function countInc (n)
        n = n or 1
        globalCounter = globalCounter + n
end
```

Expressões e Comandos

- **function:**
  - Podem retornar mais de um valor;

Exemplo de Código

```
function calcMax (list)
        local mi = 1
        local m = list[mi]
        for i = 1, #list do
                if list[i] > m then
                        mi = i; m = list[i]
                end
        end
        return m, mi
end
```

Exemplo de Saída

```
print(calcMax({8, 10, 23, 12, 5}))        → 23    3
```

Expressões e Comandos

- **function:**
  - Funções podem ser **variadic**, assim ter *varargs* como parâmetro, basta colocar **(...)** como parâmetro de entrada;

Exemplo de Código

```
function add (...)
        local s = 0
        for _, v in ipairs{...} do
                s = s + v
        end
        return s
end
```

Expressões e Comandos

- **function:**
  - Como mencionado anteriormente, Funções são Valores de Primeira Classe com Escopo Léxico Próprio;
  - Sendo Valores de Primeira Classe, podemos armazenar Funções em Variáveis (Locais ou Globais), em Tables ou passar como argumentos de outras funções e retorna-las como resultados de Funções;
  - Tendo Escopo Léxico Próprio, podemos acessar variáveis de funções anexas, sendo assim, Lua também possui cálculo Lambda;

Expressões e Comandos

- **function:**

Exemplo de Código

```
a = {p = print}          -- 'a.p' se refere a função 'print'
a.p("Hello World")       → Hello World
print = math.sin         -- 'print' agora se refere a função seno
a.p(print(1))            → 0.8414709848079
math.sin = a.p           -- 'math.sin' agora se refere a função 'print'
math.sin(10, 20)         → 10  20

teste = function (x) return 2*x end
teste(2)                 → 4
```

Expressões e Comandos

- **function:**

Exemplo de Código

```
redes = {
        {nome = "grauna", IP = "210.26.30.34"},
        {nome = "arraial", IP = "210.26.30.23"},
        {nome = "lua", IP = "210.26.23.12"},
        {nome = "derain", IP = "210.26.23.20"},
}

table.sort(redes, function (a, b) return (a.nome > b.nome) end)
```

Modularização

Modularização

- As Correspondências de Parâmetros das funções são **Posicionais**;
- O Mecanismo de Passagem é por **Cópia** e por **Referência**;
- O Momento de Passagem é **Normal**;
- Por meio de *Lazy Tables* é possível a *Lazy Evaluation*;

Modularização

- **Módulos** são carregados por meio da função **require**;
- **require** cria e retorna uma **table**;
- Tudo que é exportado pelo módulo, como funções e constantes, são definidas dentro da **table**, esta funcionando como um **namespace**;
- Todas as bibliotecas padrões são módulos;

Exemplo de Código	
local m = require "math" print(type(m.sin(3.14)))	**-- local m = require("math")** **→ number**

Modularização

- O interpretador já faz um pré-carregamento de todas as bibliotecas padrões;
- Podemos também fazer a requisição de uma função específica;

Exemplo de Código	
local f = **require** "mod".foo	**– (require(“mod”)).foo**

Polimorfismo

Polimorfismo

- **Coerção:**
  - Coerção é realizada dependendo das operações realizadas em tempo de execução;

Exemplo de Código

```
n = 1  s = "2"

newN = s + n
newS = n .. s

print(newN)          → 3
print(type(newN))    → number
print(newS)          → 12
print(type(newS))    → string
```

Polimorfismo

- **Sobrecarga:**
  - É possível realizar a sobrecarga de métodos de *metatables*;
  - Os metamétodos mencionados anteriormente e vários outros podem ser sobrescritos, e alguns deles como o **__add**, por exemplo, sobrecarregaria o operador binário **+**.

Polimorfismo

- **Sobrecarga:**

Exemplo de Código

```
mt = setmetatable({4, 4}, {
        __add = function(mt, nt)                    -- Sobrecarga do
                for i = 1, table.maxn(mt) do        -- Operador "+"
                        mt[i] = mt[i] + nt[i]
                end
                return mt
        end
}
```

Polimorfismo

- **Paramétrico:**
  - Como Lua possui tipagem dinâmica e não há necessidade de indicar os tipos dos parâmetros de entrada e saída, a própria linguagem realiza este polimorfismo;

Polimorfismo

- **Inclusão:**
  - Embora Lua não seja uma linguagem **OO**, podemos **CRIAR CLASSES** em **LUA**, um prazer de uma linguagem POG (Programação Orientada a Gambiarra);
  - Como podermos implementar Classes em Lua, também podemos também implementar **Heranças**;
  - Lua suporta **Herança Múltipla**;

Implementando uma Classe

- *Tables* são Objetos e como objetos, *Tables* possuem estados e identidades (*self*);
- Podemos implementar Classes por ***Tables*** e ***Metatables***;

Implementando uma Classe - Table

- Para implementar, por exemplo, um Monstro em um jogo, poderíamos fazer isso:

Exemplo de Código

```
monstro = { nome = “caozinho”, vida = 8001, ataque = 8001, defesa = 8001}
```

Implementando uma Classe - Table

- Mas se quisermos criar várias instâncias do mesmo, com os mesmos atributos, teríamos que fazer varias variáveis com as mesmas atribuições;

Exemplo de Código

```
monstro_1 = { nome = "caozinho", vida = 8001, ataque = 8001,
defesa = 8001}

monstro_2 = { nome = "caozinho", vida = 8001, ataque = 8001,
defesa = 8001}

monstro_3 = { nome = "caozinho", vida = 8001, ataque = 8001,
defesa = 8001}

monstro_4 = { nome = "caozinho", vida = 8001, ataque = 8001,
defesa = 8001}
```

Implementando uma Classe - Table

- Para facilitar, poderíamos criar uma função que retornasse estas *Tables*:

Exemplo de Código

```
function CriarMonstro ()
    local mob = { nome = "caozinho", vida = 8001, ataque = 8001,
defesa = 8001}
    return mob
else
```

- Com isso, para criar um novo Monstro, bastaria fazer uma variável receber o resultado da função.

Implementando uma Classe - Table

- Para criar um método para uma *Table* que criamos, basta colocar o nome dela com um **“.”** antes do nome da função:

Exemplo de Código

```
local pessoa = {nome = “Fulano”}

pessoa.HelloWorld = function()
      print(“Hello World”)
end

pessoa.HelloWorld()                    → Hello World
```

Implementando uma Classe - Table

- Se quisermos que o nome da pessoa seja escrito na tela, podemos fazer desta forma:

Exemplo de Código

```
local pessoa = {nome = "Fulano"}

pessoa.HelloWorld = function(aux)
      print(aux.nome .. " diz, Hello World")
end

pessoa.HelloWorld(pessoa)          → Fulano diz, Hello World
```

Implementando uma Classe - Table

- Para simplificamos, podemos usar o operador **":"** para chamar o método:

Exemplo de Código

```
local pessoa = {nome = "Fulano"}

pessoa.HelloWorld = function(aux)
      print(aux.nome .. " diz, Hello World")
end

pessoa:HelloWorld()                    → Fulano diz, Hello World
```

- O operador **":"** faz com que ***pessoa*** seja passada como o primeiro argumento;

Implementando uma Classe - Table

- Como se não fosse o bastante, ainda podemos utilizar o operador **":"** para definirmos o método:

Exemplo de Código

```
local pessoa = {nome = "Fulano"}

pessoa:HelloWorld = function()
      print(self.nome .. " diz, Hello World")
end

pessoa:HelloWorld()                    → Fulano diz, Hello World
```

- O operador **":"** faz com que ***pessoa*** possua agora o método **HelloWorld** em sua estrutura e a função tem um primeiro parâmetro "**SECRETO**" chamado *self*.

Implementando uma Classe - Table

- Voltando para a Criação do Monstro teremos, de maneira mais prática:

Exemplo de Código

```
Monstro = {}

function Monstro:Criar ()
    local mob = { nome = "caozinho", vida = 8001, ataque = 8001,
defesa = 8001}
    return mob
end

monstro_1 = Monstro:Criar()
```

Implementando uma Classe - Table

- Para inserirmos métodos para esta *Table*:

Exemplo de Código

```
Monstro = {}

function Monstro:Criar ()
    local mob = { nome = "caozinho", vida = 8001, ataque = 8001,
defesa = 8001}

    function mob:Morder()
        print(self.nome .. " mordeu você.")
    return mob
end

monstro_1 = Monstro:Criar()
monstro_1:Morder()                    → caozinho mordeu você.
```

Implementando uma Classe - Metatable

- Para implementar utilizando **Metatable** precisamos utilizar o metamétodo **__index;**
- A criação de Classes se dará por meio de **Herança**;
  - Se tivermos dois objetos **A** e **B**, sendo **A** uma classe, faremos com que **A** "herde" **B**;

Exemplo de Código

```
setmetatable(A, {__index = B}}
```

  - Com isso **A** procura em **B** qualquer operação que não possua;

Implementando uma Classe - Metatable

- Um exemplo para Conta de Banco:

Exemplo de Código

```
local mt = {__index = Conta }

function Conta.Criar (o)
      o = o or {}                    -- Cria uma Table vazia caso não
      setmetatable(o, mt)            -- tenha sido passada
      return o
end

a = Conta.Criar{saldo = 0}           -- Faz "a" herdar "Conta"
a:Depositar(100.00)                  -- Utilizando método de Conta
```

Implementando uma Classe - Metatable

- Para o mesmo exemplo de deposito, também poderíamos usar desta forma:

Exemplo de Código

```
local mt = {__index = Conta }

function Conta.Criar (o)
      o = o or {}                    -- Cria uma Table vazia caso não
      setmetatable(o, mt)            -- tenha sido passada
      return o
end

a = Conta.Criar{saldo = 0}
Conta.Depositar(a, 100.00)           -- Mesmo resultado do anterior
```

Implementando uma Classe - Metatable

- Para utilizarmos o operador “:” para criação de classes e utilizando **self** podemos fazer desta forma:

Exemplo de Código

```
function Conta:Criar (o)
    o = o or {}
    self.__index = self
    setmetatable(o, self)
    return o
end

a = Conta:Criar()
```

- Além de herdar métodos, os valores também são herdados;

Implementando uma Classe - Metatable

- Para herdarmos, podemos criar outra "Classe" herdando de uma já criada:

Exemplo de Código

```
ContaEspecial = Conta:Criar()

s = ContaEspecial:Criar{limite = 1000.00}
```

- Com isso temos que: **ContaEspecial** herda de **Conta**, logo **s** herda de **ContaEspecial** que por sua vez herda de **Conta**;
- Se formos utilizar o método **s:Depositar(100.00)**, não será encontrado **Depositar** em **s**, então será olhado em **ContaEspecial**, como também não se encontra aqui, será olhado em **Conta**, onde será encontrada a implementação Original, sendo parecido com Java;

Implementando uma Classe - Metatable

- Com isso, podemos reescrever métodos:

Exemplo de Código

```
function Conta:Depositar (v)
        self.saldo = self.saldo + v
end

function ContaEspecial:Depositar (v)
        self.saldo = self.saldo + v + (0.1 * v)
end
```

- Neste caso, se usarmos **Depositar** em uma **Herança** de **ContaEspecial** o método **ContaEspecial:Depositar** que será executado;

Implementando uma Classe - Metatable

- Nem tudo são flores, para Herança Múltipla temos:

Exemplo de Código

```
-- Procura por K na lista de Tables "plist"
local function pesquisar (k, plist)
    for i = 1, #plist do
        local v = plist[i][k]
        if v then return v end
    end
end

---- CONTINUA
```

Implementando uma Classe - Metatable

Exemplo de Código

```
function criarClasse (...)
    local c = {}                        -- Nova Classe
    local plist = {...}                 -- Lista de Heranças
    -- Busca pelos métodos para inserir em "c"
    setmetatable(c, {__index = function (t, k)
                                    return pesquisar(k, plist)  end})
    c.__index = c      -- Prepara "c" para virar uma metatable

    function c:Criar (o)    -- Define o novo construtor dessa classe
        o = o or {}
        setmetatable(o, c)
        return o
    end
    return c                    -- Retorna a nova classe
end
```

Implementando uma Classe - Metatable

- Tenhamos, além da "Classe" **Conta** já mencionada, esta nova "Classe" chamada **Nomear**:

Exemplo de Código

```
Nomear = {}
function Nomear:getNome ()
    return self.nome
end

function Nomear:setNome (n)
    self.nome = n
end
```

Implementando uma Classe - Metatable

- Para que possamos fazer a **Herança Múltipla** de **Nomear** e **Conta**, basta utilizar a função **criarClasse** mencionada;

Exemplo de Código

```
ContaNomeada = criarClasse (Conta, Nomear)

contaTeste = ContaNomeada:Criar{name = "Jordan"}
print(contaTeste:getNome())      → Jordan
```

Exceções

Exceções

- Não há Exceções, mas há erros;
- Não é necessário fazer o tratamento de erro em Lua, elas já são realizadas pela aplicação do programa;
- Caso queria realizar o tratamento de erros, pode ser utilizada a função **pcall** para encapsular o código;
- **pcall** retorna **true** caso não haja nenhum erro ou **false** e a **mensagem deixada** (referida pela função **error**) caso tenha acontecido algum erro;

Exceções

- Exemplo de utilização:

Exemplo de Código

```
local status, err = pcall(function ()
        <alguma instrução>
        if condicao_inesperada then error() end
        <alguma instrução>
end)

if status then              -- Sem erros no código encapsulado
        <instruções regulares>
else                        -- Houve erro, tome as medidas apropriadas
        <instruções de tratamento de erro>
end
```

Concorrência

Concorrência

- Realizada por meio de *coroutines*;
- Uma *coroutine* é similar a uma *thread* (no sentido de *multithreading*): é uma linha de execução, com sua própria pilha, suas próprias variáveis locais e seu próprio ponteiro de instrução, compartilhando variáveis globais e quase tudo com outras *coroutines*;
- Uma *coroutine* possui o tipo **thread**, como descrito anteriormente em **Valores e Tipos de Dados**;

Concorrência

- A principal diferença entre *coroutines* e multithread é:
  - Um programa *multithread* executa várias *threads* em paralelo;
  - *Coroutines* são colaborativas, ou seja, um programa com *coroutines* está executando apenas uma de suas *coroutines*, e essa *coroutine* em execução suspende sua execução somente quando a solicita explicitamente;
- Uma *coroutine* pode ter quatro status: *suspended, running, normal* e *dead;*

Concorrência

- Todas as funções relacionadas a *coroutines* estão na table ***coroutine***;
- **coroutine.create(f)**: cria uma nova *coroutine*, recebe um único argumento **f** que é a função que esta irá executar;
- **coroutine.status(c):** retorna uma *string* com o status da *coroutine* **c;**
  - Quando uma *coroutine* é criada, seu status inicial é de **suspended**;

Concorrência

- **coroutine.resume(c):** inicia (ou retoma) a execução de uma *coroutine*, alterando seu status de **suspended** para **running**;
  - Quando uma *Coroutine* termina sua execução, ela tem seu status alterado para **dead**;
- **coroutine.yield():** faz com que a *coroutine* se **auto suspenda** enquanto está em execução;
  - Com isso, durante sua execução, uma *coroutine* pode mudar seu status de **running** para **suspended**;
  - Os valores colocados como parâmetros são retornados para a função **coroutine.resume**, também retornando qualquer argumento extra passado para **resume**;

Concorrência

Exemplo de Código

```
hiprint = coroutine.create  (function () print("hi") end)

print(type(hiprint))                    → thread

print(coroutine.status(hiprint))        → suspended

coroutine.resume(hiprint)               → hi

print(coroutine.status(hiprint))        → dead
```

Avaliação da Linguagem

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Parcial	Não	Não	Sim
Aprendizado	Parcial	Não	Não	Não
Eficiência	Parcial	Sim	Sim	Parcial
Portabilidade	Sim	Não	Não	Sim
Método de Projeto	Multiparadigma	Estruturado	Estruturado e OO	OO
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	**Sim**	**Sim**	**Sim**	**Parcial**
Confiabilidade	**Lua é a principal Linguagem de Script para Jogos atualmente. Sendo uma Biblioteca em C, pode ser compilada em qualquer plataforma com um Compilador C ou C++.**			
Aprendizado				
Eficiência				
Portabilidade	Sim	Não	Não	Sim
Método de Projeto	Multiparadigma	Estruturado	Estruturado e OO	OO
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	**Parcial**	**Não**	**Não**	**Sim**
Aprendizado	**Mesmo possuindo Coletor de Lixo, sua tipagem é dinâmica, isso faz com que o Programador possa cometer vários erros durante a construção do código.**			
Eficiência				
Portabilidade				
Método de Projeto	Multiparadigma	Estruturado	Estruturado e OO	OO
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Parcial	Não	Não	Sim
Aprendizado	**Parcial**	**Não**	**Não**	**Não**
Eficiência	**Mesmo havendo vários mecanismos que facilitem a aprendizagem, como a não necessidade de se preocupar com a memória, entre outros, caso seja necessário uma aplicação mais efetiva, os programas passam a ser mais complicados, como a criação de Classes e Heranças.**			
Portabilidade				
Método de Projeto				
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Parcial	Não	Não	Sim
Aprendizado	Parcial	Não	Não	Não
Eficiência	**Parcial**	**Sim**	**Sim**	**Parcial**
Portabilidade				
Método de Projeto	Multiparadigma	Estruturada	OO	OO
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta

Assim como em Java, o Coletor de Lixo acaba reduzindo um pouco a eficiência.

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Parcial	Não	Não	Sim
Aprendizado	Parcial	Não	Não	Não
Eficiência	Parcial	Sim	Sim	Parcial
Portabilidade	**Sim**	**Não**	**Não**	**Sim**
Método de Projeto	**Pode ser compilada em qualquer plataforma que tenha um Compilador de C padrão.**			
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	Sim	Sim	Sim	Parcial
Custo	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Parcial	Não	Não	Sim
Aprendizado	Parcial	Não	Não	Não
Eficiência	Parcial	Sim	Sim	Parcial
Portabilidade	Sim	Não	Não	Sim
Método de Projeto	**Multiparadigma**	**Estruturado**	**Estruturado e OO**	**OO**
Evolutibilidade	**Lua pode ser programada como Procedural, Scripting, Funcional, Orientada a Objetos. Seu paradigma é "TUDO".**			
Reusabilidade				
Integração				
Custo	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Parcial	Não	Não	Sim
Aprendizado	Parcial	Não	Não	Não
Eficiência	Parcial	Sim	Sim	Parcial
Portabilidade	Sim	Não	Não	Sim
Método de Projeto	Multiparadigma	Estruturado	Estruturado e OO	OO
Evolutibilidade	**Sim**	**Não**	**Parcial**	**Sim**
Reusabilidade	**Por causa da tipagem dinâmica, fica difícil manter o código se este não estiver muito bem documentado, pois o código ao passar entre programadores diferentes, estes não vão saber o que cada varíavel deveria esperar.**			
Integração				
Custo				

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Parcial	Não	Não	Sim
Aprendizado	Parcial	Não	Não	Não
Eficiência	Parcial	Sim	Sim	Parcial
Portabilidade	Sim	Não	Não	Sim
Método de Projeto	Multiparadigma	Estruturado	Estruturado e OO	OO
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	**Sim**	**Parcial**	**Sim**	**Sim**
Integração	**Lua oferece reuso de funções, módulos e até mesmo classes.**			
Custo	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta	Depende da Ferramenta

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Parcial	Não	Não	Sim
Aprendizado	Parcial	Não	Não	Não
Eficiência	Parcial	Sim	Sim	Parcial
Portabilidade	Sim	Não	Não	Sim
Método de Projeto	Multiparadigma	Estruturado	Estruturado e OO	OO
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	**Sim**	**Sim**	**Sim**	**Parcial**
Custo	**Lua pode ser acoplada em diversas linguagens de programação, como o próprio C e Python.**			

Avaliação da Linguagem – Lua

Critérios Gerais	Lua	C	C++	Java
Aplicabilidade	Sim	Sim	Sim	Parcial
Confiabilidade	Parcial	Não	Não	Sim
Aprendizado	Parcial	Não	Não	Não
Eficiência	Parcial	Sim	Sim	Parcial
Portabilidade	Sim	Não	Não	Sim
Método de Projeto	Multiparadigma	Estruturado	Estruturado e OO	OO
Evolutibilidade	Sim	Não	Parcial	Sim
Reusabilidade	Sim	Parcial	Sim	Sim
Integração	**Depende de qual ferramenta principal será utilizada.**			
Custo	**Depende da Ferramenta**	**Depende da Ferramenta**	**Depende da Ferramenta**	**Depende da Ferramenta**

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Escopo	Sim	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	Sistema	Programador	Programador	Sistema
Persistência dos Dados	LuaSQL, Serialização	Biblioteca de Funções	Biblioteca de Classes e Funções	JDBC, Biblioteca de Classes, Serialização
Passagem de Parâmetros	Lista Variável, default, por Valor e por Referência	Lista Variável e por Valor	Lista Variável, default, por Valor e por Referência	Lista Variável, default, por Valor e por Cópia de Referência

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Escopo	**Sim**	**Sim**	**Sim**	**Sim**
Expressões e Comandos	**O escopo é definido em blocos terminados com a palavra reservada END. O escopo padrão é Global.**			
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	Sistema	Programador	Programador	Sistema
Persistência dos Dados	LuaSQL, Serialização	Biblioteca de Funções	Biblioteca de Classes e Funções	JDBC, Biblioteca de Classes, Serialização
Passagem de Parâmetros	Lista Variável, default, por Valor e por Referência	Lista Variável e por Valor	Lista Variável, default, por Valor e por Referência	Lista Variável, default, por Valor e por Cópia de Referência

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Escopo	Sim	Sim	Sim	Sim
Expressões e Comandos	**Sim**	**Sim**	**Sim**	**Sim**
Tipos Primitivos e Compostos	**Lua oferece varias expressões e comandos como demonstrado no Seminário.**			
Gerenciamento de Memória	Sistema	Programador	Programador	Sistema
Persistência dos Dados	LuaSQL, Serialização	Biblioteca de Funções	Biblioteca de Classes e Funções	JDBC, Biblioteca de Classes, Serialização
Passagem de Parâmetros	Lista Variável, default, por Valor e por Referência	Lista Variável e por Valor	Lista Variável, default, por Valor e por Referência	Lista Variável, default, por Valor e por Cópia de Referência

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Escopo	Sim	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	**Sim**	**Sim**	**Sim**	**Sim**
Gerenciamento de Memória	**Lua oferece 8 tipos: nil, boolean, number, string, function, userdata, thread e table.**			
Persistência dos Dados	LuaSQL, Serialização	Biblioteca de Funções	Biblioteca de Classes e Funções	JDBC, Biblioteca de Classes, Serialização
Passagem de Parâmetros	Lista Variável, default, por Valor e por Referência	Lista Variável e por Valor	Lista Variável, default, por Valor e por Referência	Lista Variável, default, por Valor e por Cópia de Referência

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Escopo	Sim	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	**Sistema**	**Programador**	**Programador**	**Sistema**
Persistência dos Dados	**Lua deixa a cargo do Coletor de Lixo do tipo Mark-Sweep Incremental**			
	Serialização	Funções	Classes e Funções	de Classes, Serialização
Passagem de Parâmetros	Lista Variável, default, por Valor e por Referência	Lista Variável e por Valor	Lista Variável, default, por Valor e por Referência	Lista Variável, default, por Valor e por Cópia de Referência

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Escopo	Sim	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	Sistema	Programador	Programador	Sistema
Persistência dos Dados	**LuaSQL, Serialização**	**Biblioteca de Funções**	**Biblioteca de Classes e Funções**	**JDBC, Biblioteca de Classes, Serialização**
Passagem de Parâmetros	**Lua possui uma variedade de Bancos de Dados ao qual pode usar por meio da LuaSQL. Possui bibliotecas para utilizar JSON e possui Table Serializaton.**			
	Referência		Referência	Referência

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Escopo	Sim	Sim	Sim	Sim
Expressões e Comandos	Sim	Sim	Sim	Sim
Tipos Primitivos e Compostos	Sim	Sim	Sim	Sim
Gerenciamento de Memória	Sistema	Programador	Programador	Sistema
Persistência dos Dados	LuaSQL,	Biblioteca de	Biblioteca de Classes e	JDBC, Biblioteca de Classes
Passagem de Parâmetros	**Lista Variável, default, por Valor e por Referência**	**Lista Variável e por Valor**	**Lista Variável, default, por Valor e por Referência**	**Lista Variável, default, por Valor e por Cópia de Referência**

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Encapsulamento e Proteção	Parcial	Parcial	Sim	Sim
Sistema de Tipos	Sim	Não	Parcial	Sim
Verificação de Tipos	Dinâmica	Estática	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Todos	Coerção e Sobrecarga	Todos	Todos
Exceções	Não	Não	Parcial	Sim
Concorrência	Sim	Não (biblioteca de funções)	Não (biblioteca de funções)	Sim

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Encapsulamento e Proteção	**Parcial**	**Parcial**	**Sim**	**Sim**
Sistema de Tipos	**Mesmo possuindo Classes, mecanismo de Private, ainda é possível manusear bibliotecas importadas e alterar seus valores.**			
Verificação de Tipos	Dinâmica	Estática	Dinâmica	Dinâmica
Polimorfismo	Todos	Coerção e Sobrecarga	Todos	Todos
Exceções	Não	Não	Parcial	Sim
Concorrência	Sim	Não (biblioteca de funções)	Não (biblioteca de funções)	Sim

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Encapsulamento e Proteção	Parcial	Parcial	Sim	Sim
Sistema de Tipos	**Sim**	**Não**	**Parcial**	**Sim**
Verificação de Tipos	**Não é possível violar o Sistema de Tipos de Lua como em C.**		Dinâmica	Dinâmica
Polimorfismo	Todos	Coerção e Sobrecarga	Todos	Todos
Exceções	Não	Não	Parcial	Sim
Concorrência	Sim	Não (biblioteca de funções)	Não (biblioteca de funções)	Sim

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Encapsulamento e Proteção	Parcial	Parcial	Sim	Sim
Sistema de Tipos	Sim	Não	Parcial	Sim
Verificação de Tipos	**Dinâmica**	**Estática**	**Estática / Dinâmica**	**Estática / Dinâmica**
Polimorfismo	**Por ser uma Linguagem Tipada Dinamicamente, a verificação ocorre em tempo de execução.**			
Exceções	Não	Não	Parcial	Sim
Concorrência	Sim	Não (biblioteca de funções)	Não (biblioteca de funções)	Sim

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Encapsulamento e Proteção	Parcial	Parcial	Sim	Sim
Sistema de Tipos	Sim	Não	Parcial	Sim
Verificação de Tipos	Dinâmica	Estática	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	**Todos**	**Coerção e Sobrecarga**	**Todos**	**Todos**
Exceções	**Possui todos os tipos, mesmo que o Paramétrico seja feito automaticamente pela linguagem.**			
Concorrência	Sim	de funções)	de funções)	Sim

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Encapsulamento e Proteção	Parcial	Parcial	Sim	Sim
Sistema de Tipos	Sim	Não	Parcial	Sim
Verificação de Tipos	Dinâmica	Estática	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Todos	Coerção e Sobrecarga	Todos	Todos
Exceções	**Não**	**Não**	**Parcial**	**Sim**
Concorrência	**Não possui mecanismos de Exceções, mas possui mecanismo para tratamento de erro.**			

Avaliação da Linguagem – Lua

Critérios Específicos	Lua	C	C++	Java
Encapsulamento e Proteção	Parcial	Parcial	Sim	Sim
Sistema de Tipos	Sim	Não	Parcial	Sim
Verificação de Tipos	Dinâmica	Estática	Estática / Dinâmica	Estática / Dinâmica
Polimorfismo	Todos	Coerção e	Todos	Todos
Exceções	**Possui concorrência por meio de Coroutines próprias da linguagem.**			
Concorrência	**Sim**	**Não (biblioteca de funções)**	**Não (biblioteca de funções)**	**Sim**

Referências

- Manual de Referência Lua 5.2. https://www.lua.org/manual/5.2/manual.html, último acesso em 24/11/2019;
- Manual de Referência Lua 5.3. https://www.lua.org/manual/5.3/manual.html, último acesso em 24/11/2019;
- IERUSALIMSCHY, R. Programming in Lua. 4ed. Feisty Duck Digital. 2016