

# Minicurso de Lua

Diogo Krüger Souto



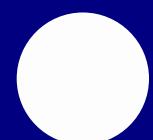
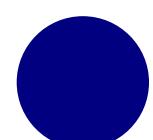


**Por que um  
minicurso sobre  
Lua?**



# Por que um minicurso sobre Lua?

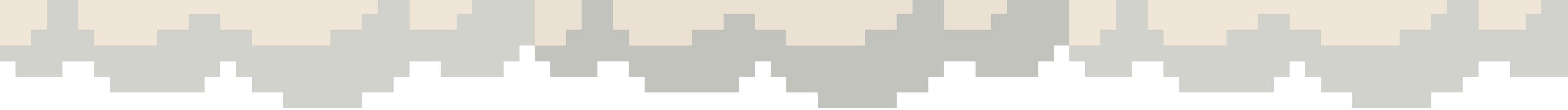
Devemos valorizar mais  
as tecnologias criadas  
no nosso próprio país



# **Por que um minicurso sobre Lua?**

Lua é uma linguagem de  
programação extremamente  
simples e poderosa



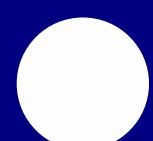
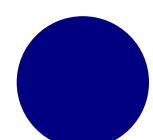


# O que esperar desse minicurso?



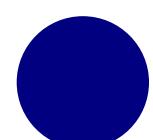
# **Depende:**

Pessoas em diferentes fases de seus estudos  
em programação poderão extrair benefícios  
distintos e subjetivos deste curso.



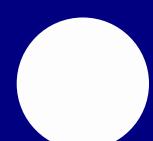
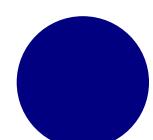
# **Aqueles no início da sua jornada:**

- Aproveitar da simplicidade da linguagem para aprender programação
- Ter a chance de entender alguns conceitos que permeiam nossa área



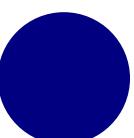
# **Aqueles com alguma experiência:**

- Obter perspectivas diferenciadas sobre a programação oferecidas pela linguagem
- Desbravar lados da programação outrora desconhecidos (scripting, embedding, etc)



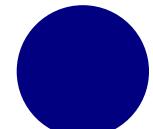
# **Todo mundo:**

- Pelo menos compreender uma tecnologia do nosso próprio país
- Talvez adicionar uma nova ferramenta ao seu arsenal :)



# O que faremos:

- Introduzir a linguagem, contando sua história e mostrando as suas características.
  - Passar pelos seus conceitos mais básicos até os mais avançados.
  - Expor como e para quê podemos utilizar esses conceitos.
  - Faremos algumas pausas para exercitar o que aprendemos.
- 





**Antes de tudo...**

De onde surgiu **Lua**?





**Lua foi criada na  
PUC-Rio em 1993**

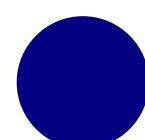
Fruto de uma parceria  
do instituto TeCGraf  
com a Petrobras frente  
à dois problemas:



Waldemar Celes, Roberto Ierusalimschy, Luiz  
Henrique de Figueiredo, respectivamente

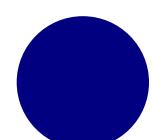
# **Antes de Lua**

**1) Engenheiros da Petrobras precisavam preparar arquivos de dados de entrada para simuladores várias vezes ao dia.**



## **Antes de Lua**

2) Precisavam também de uma interface para um gerador de relatórios configurável para perfis de litologia

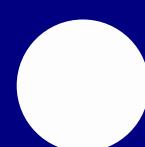
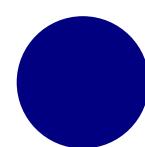


# **Antes de Lua**

**Para isso, o  
TeCGraf  
desenvolveu  
duas  
línguagens:**

**DEL**  
**(Data Entry Language)**

**SOL**  
**(Simple Object Language)**



## Código em

### DEL:

**:e** define uma entidade com campos e seus valores

**:p** define restrições aos valores de uma entidade

## Antes de Lua

```
:e      gasket
mat    s
m      f      0
y      f      0
t      i      1
"gasket properties"
# material
# factor m
# settlement stress
# facing type

:p
gasket.m>30
gasket.m<3000
gasket.y>335.8
gasket.y<2576.8
```

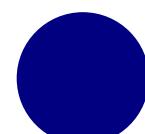
Código em  
**SOL:**

## Antes de Lua

```
type @track {x:number, y:number = 23, z = 0}
```

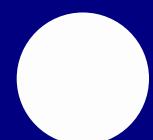
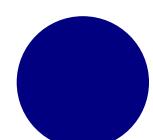
define um tipo '**track**', com os atributos numéricos **x** e **y**,  
e também um atributo sem tipo **z**.

**y** e **z** possuem valores padrão.



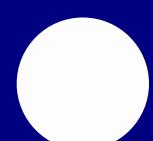
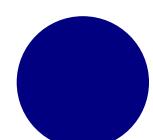
## **Antes de Lua**

Em 1993 as duas linguagens foram combinadas, culminando na primeira versão de **Lua!**



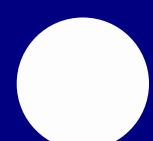
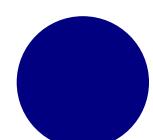


**Por que aprender  
Lua hoje em dia?**



# **Por que aprender Lua hoje em dia?**

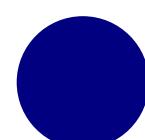
Lua é uma das linguagens que  
mais cresceu nos últimos anos



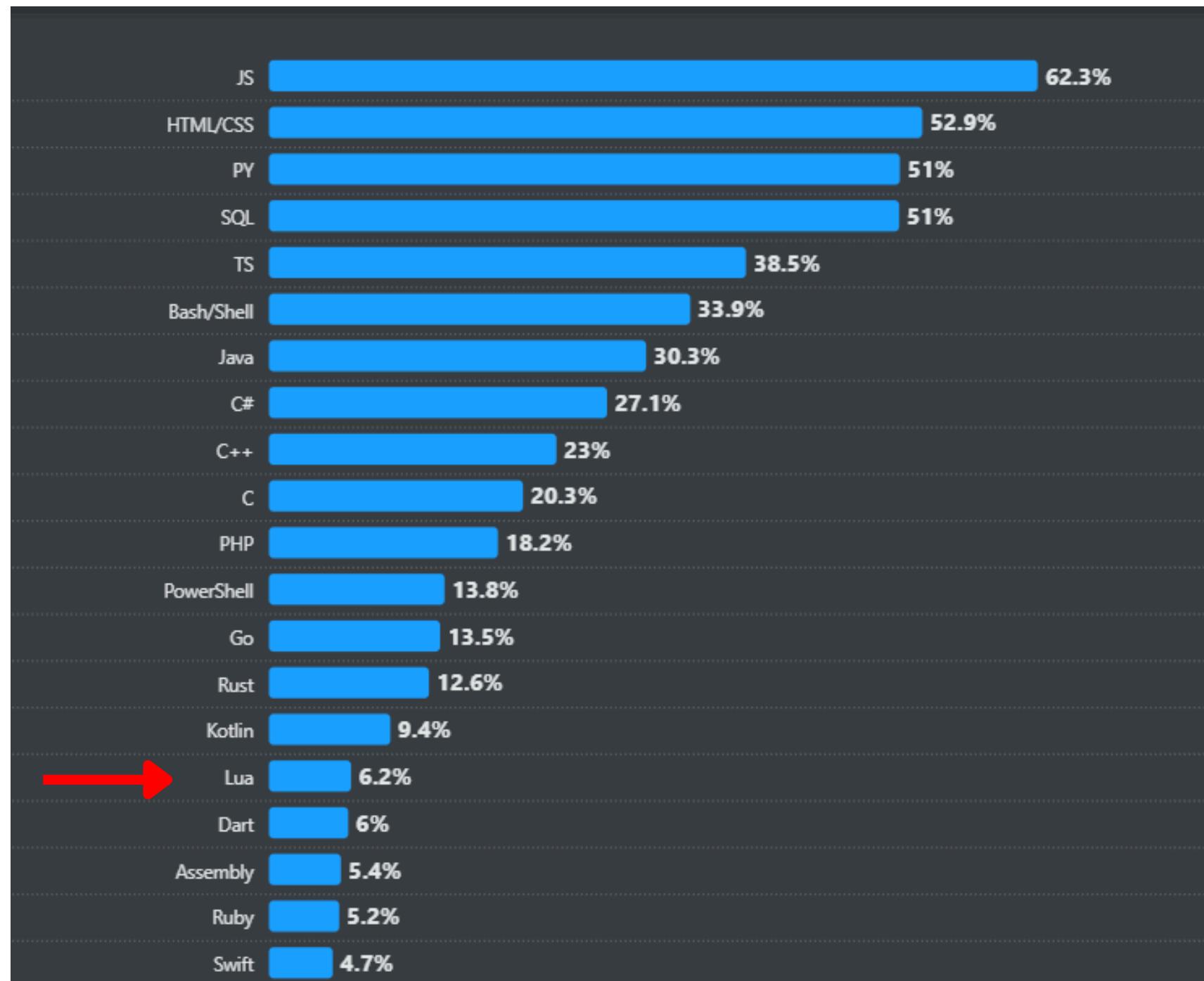
**“Conhecida por sua utilidade no desenvolvimento de jogos, Lua é amplamente utilizada no domínio de aplicações e no desenvolvimento de jogos.**

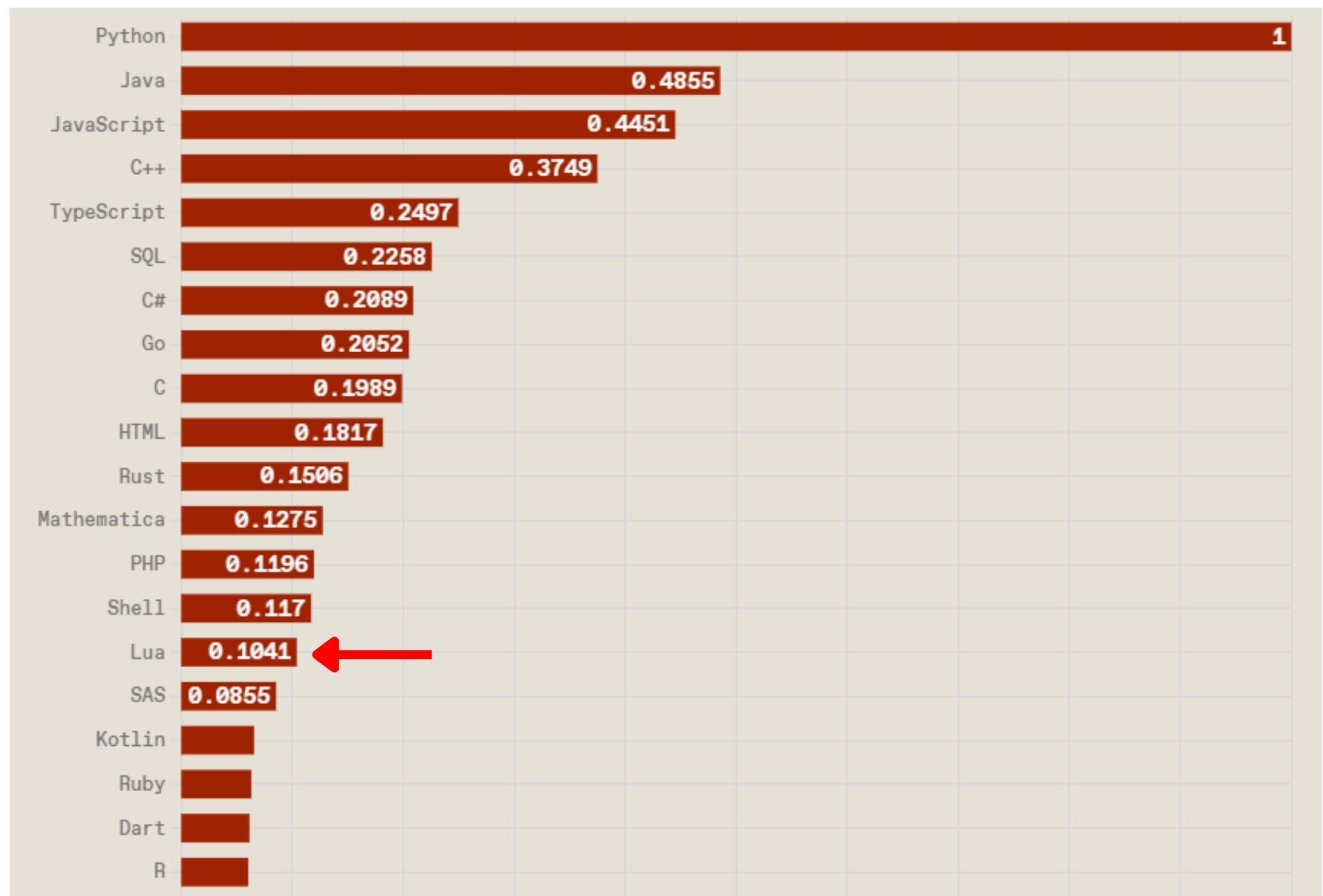
Notavelmente, ela também é usada como uma linguagem de script simples em cenários que variam de jogos a desenvolvimento de aplicações e à Internet das Coisas (IoT).”

01 HCL	56.1%
02 Rust	50.5%
03 TypeScript	37.8%
04 Lua	34.2%
05 Go	28.3%
06 Shell	27.7%
07 Makefile	23.7%
08 C	23.5%
09 Kotlin	22.9%
10 Python	22.5%



**“Em quais linguagens de programação, script e markup você desenvolveu bastante no ano passado e em quais você quer trabalhar no próximo ano?”**

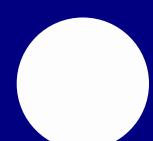
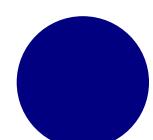






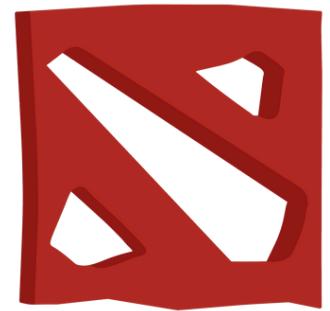
# **Por que aprender Lua hoje em dia?**

É aplicável em diferentes  
áreas, como:





The  
**SIMS**



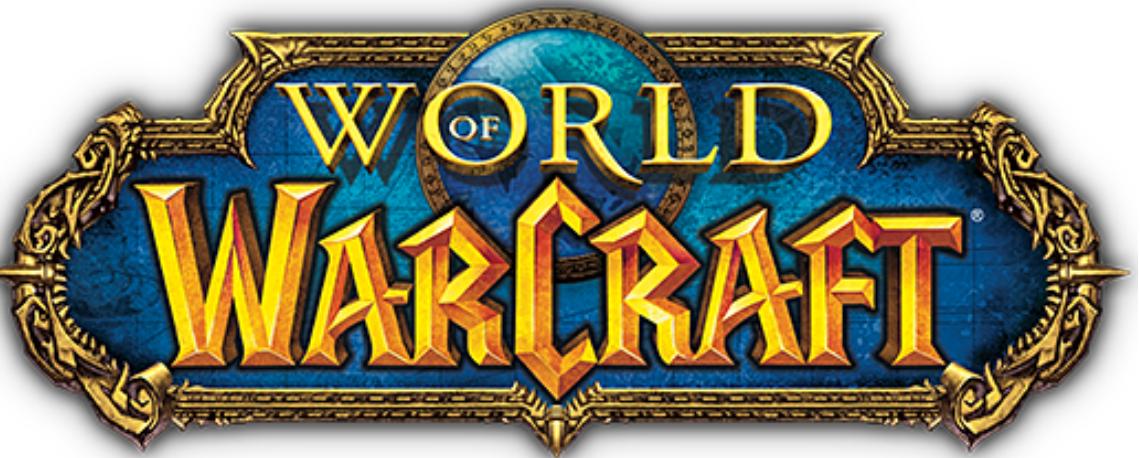
DOTA 2



Baldur's Gate



# Jogos

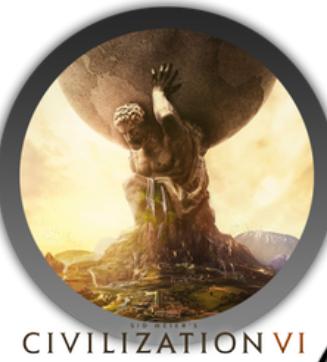


grand  
theft  
auto  
ONLINE  
role play

L.A. Noire

FARCRY

garry's mod





# Jogos

- primeiro jogo Triple-A a utilizar Lua, em 1998
- lançado pela LucasArts

# Jogos



→ usa uma versão  
modificada de Lua  
chamada “Luau”  
(open source)

# Jogos



- exemplo mais recente
- feito inteiramente em Lua  
com Löve2D
- ganhou diversas premiações  
de melhor jogo do ano

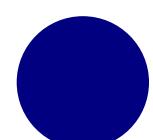
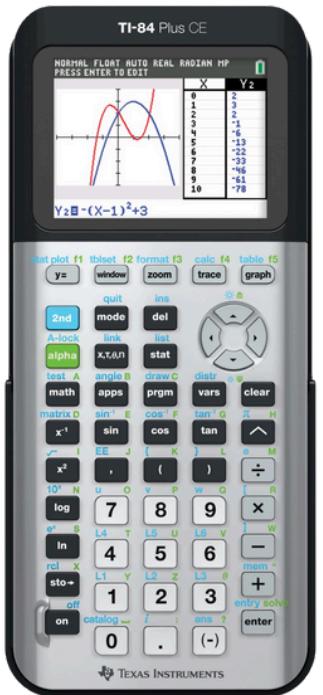


**Lua não é apenas  
uma “linguagem  
de brinquedo”!**



# Embarcados

- Satélites da ISRO
- Câmeras da Nikon
- Teclados da Logitech
- Roteadores da CISCO
- Televisões da Samsung
- Calculadoras da Texas Instruments
- Painéis de carro da Mercedes e Volvo

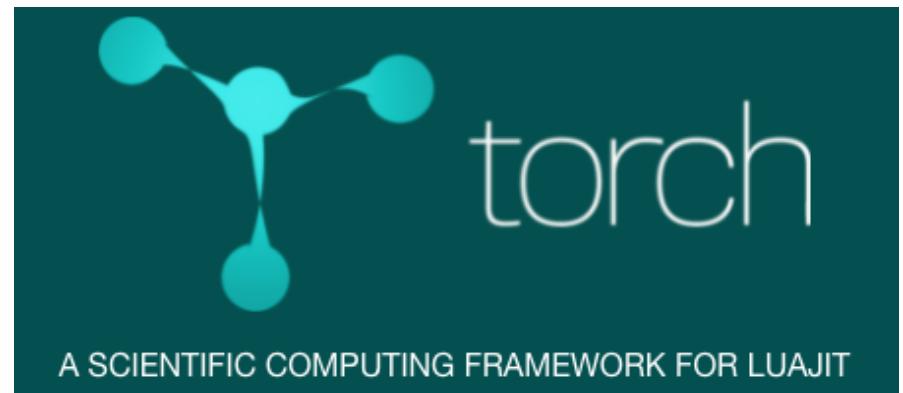




**Pela natureza dos mercados  
desses dois nichos, por  
muito tempo, a comunidade  
esteve separada e Lua  
“operou nas sombras”**



# Software

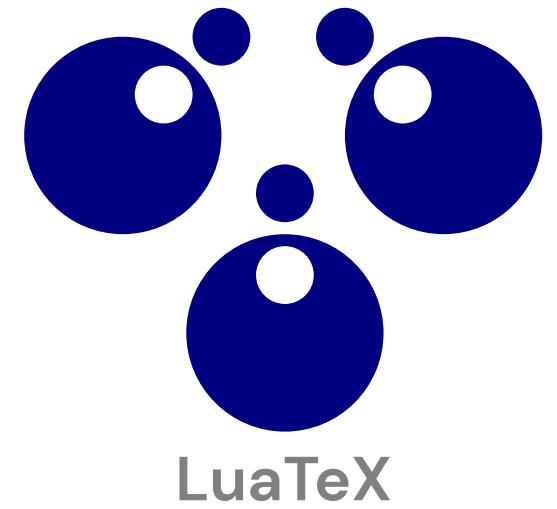


→ antecessor do PyTorch



→ provê scripting para  
aplicações Qt

# Software

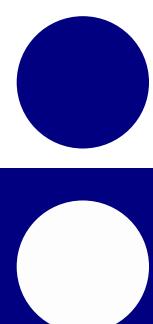


→ extensão para typesetting  
de texto para TeX



“servidor leve de HTTP e proxy reverso”

→ permite configuração  
utilizando **Lua**



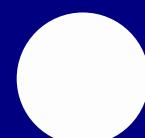


# Software

→ cache, banco de dados leve  
ou sistema de mensagens



→ **Lua** pode ser usada  
para configurar o OBS



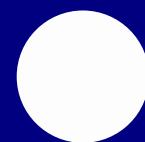
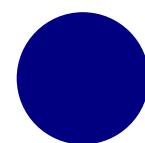
# Software



→ mais de 1 milhão de linhas

de código **Lua**

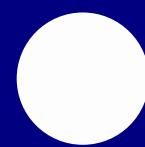
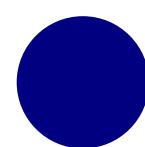
→ Framework de criação  
de jogos que utiliza **LuaJIT**





# Software

- Editor de texto configurável completamente em **Lua**.
- Vasta gama de plugins que podem o tornar em uma IDE “que tem a sua cara”



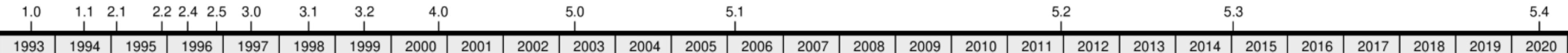


# Vamos começar

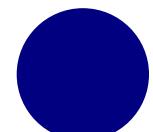


# Características de Lua

Lua está na versão 5.4.8



5.4 → a versão  
.8 → o release

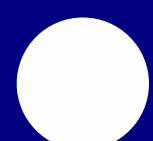
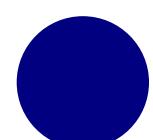


# **Características de Lua**

**Lua está na versão 5.4.8**

**Versões diferentes são  
realmente diferentes,  
geralmente alterando a  
API C e a VM.**

**Releases são designados  
apenas para correções  
de bugs e possuem o  
mesmo manual.**



# Características de Lua

Seu código fonte possui 30 mil linhas de código ANSI C

O interpretador Lua construído com todas as bibliotecas Lua padrão ocupa 279K

A biblioteca Lua para integração com C ocupa 464K



# **Características de Lua**

**Lua é portável para diversos sistemas operacionais**

Lua consegue rodar em plataformas como Windows, MacOS, Posix (Linux, BSD), iOS, Android, Arduino, Raspberry Pi, PSP, Playstation, Nintendo Switch e também dentro do kernel do Linux.



# Características de Lua

Código .lua

ex.:

programa.lua

É chamado o  
interpretador

"lua programa.lua"

A linguagem é  
interpretada

compilado

bytecode  
lua

é levado para

Execução  
do código

Máquina  
virtual

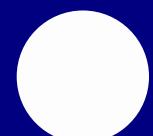
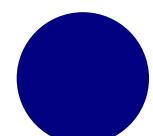
interpreta e  
executa o  
código  
instrução por  
instrução

# Características de Lua

Lua pode ser tanto uma  
linguagem de **embedding**  
quanto **extending**

Embedding: Lua pode ser  
utilizada com **biblioteca** para  
extender uma aplicação (C, por  
exemplo)

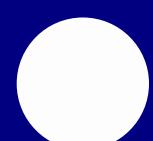
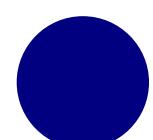
Extending: Outras linguagens de  
programação (como C) podem  
implementar funções em Lua.  
**(Parecido com Python)**



# Características de Lua

Lua é multi-paradigma:

- É **imperativa e procedural**, tal como C e Python.
- Através de seus mecanismos, pode implementar **Orientação a Objeto**.
- Suporta **Programação Funcional** pela forma como lida com funções.



# Características de Lua

Lua tem tipagem **dinâmica** e **forte\***

→ Não se declara tipos de variáveis, a linguagem que lida com isso.

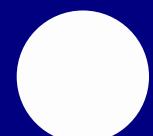
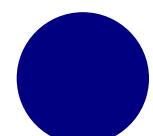
Variáveis podem mudar de tipo ao longo da execução

→ Lua não converte tipos automaticamente de forma implícita, você que deve fazer isso.

→ às vezes  
converte sim

# Características de Lua

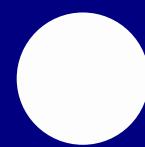
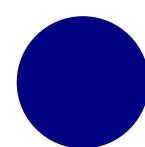
Lua é Garbage Collected



# Características de Lua

Lua é uma linguagem  
de **scripting**:

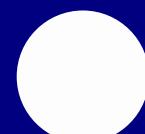
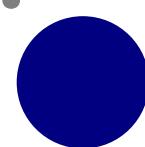
- Código-fonte é executado diretamente
- É feita para automatizar tarefas
- Mais fácil de integrar com sistemas existentes



# Características de Lua

Lua é uma linguagem  
de **scripting**:

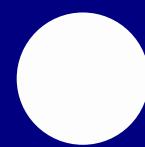
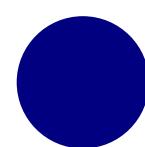
- Linguagem de **sistemas** vs.  
linguagem de **scripting**  
A linguagem de **sistemas**  
implementa a parte  
“complexa” e que “não muda  
muito”, como algoritmos  
pesados e a lógica principal.



# Características de Lua

Lua é uma linguagem  
de **scripting**:

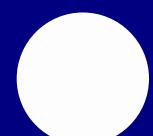
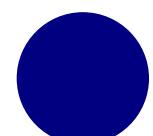
- Linguagem de **sistemas** vs.  
linguagem de **scripting**  
Já a linguagem de **scripting**  
provê uma comunicação entre  
essas linguagens, “colando”  
tudo e sendo flexível.



# **Características de Lua**

Analogia com um filme:

A linguagem de sistemas é como o set de filmagens ou, os equipamentos para gravação, aquilo que muda pouco de filme para filme. Já a linguagem de script é, realmente, como o script do filme, definindo onde as cenas irão entrar, quais personagens falarão e quando e etc.



Lua é uma das linguagens de **script** mais rápidas



Benchmark de Fibonacci com recursão:

Interpreted, dynamically typed			
Language	RunTime	Run	Ext
Escript	28.380	escript fib.es	es
Scheme	102.887	guile fib.scm	scm
Php	157.312	php fib.php	php
Lua	203.702	lua fib.lua	lua
Ruby	393.625	ruby fib.rb	rb
Python	423.427	python fib.py	py
Janet	479.663	janet ./fib.janet	janet
Perl	1490.416	perl fib.pl	pl
Raku	1672.015	rakudo fib.raku	raku
Tcl	2230.883	tclsh fib.tcl	tcl
R	2575.249	R -f fib.r	r

Sua implementação compilada em tempo de execução é mais rápida ainda!

Lua Jit	37.837
Python (PyPy)	54.078

# Características de Lua

Lua possui 22 keywords

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



# Características de Lua

Lua possui 22 keywords

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



# Características de Lua

Lua possui 33 tokens

+	-	*	/	%	^	#
&	~		<<	>>	//	
==	~=	<=	>=	<	>	=
(	)	{	}	[	:::	
;	:	,	.	..	...	



# Características de Lua

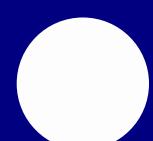
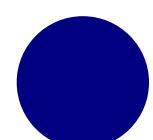
Lua possui 33 tokens

+	-	*	/	%	^	#
&	~		<<	>>	//	
==	~=	<=	>=	<	>	=
(	)	{	}	[	:::	
;	:	,	.	...	...	

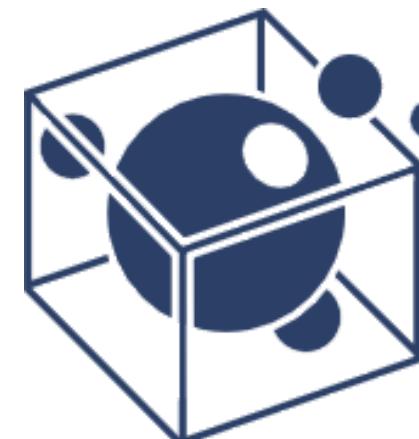


# **Características de Lua**

Lua foi feita com matemáticos  
e engenheiros em mente, isso  
explica sua sintaxe

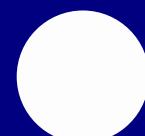
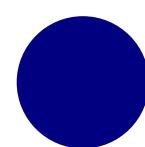
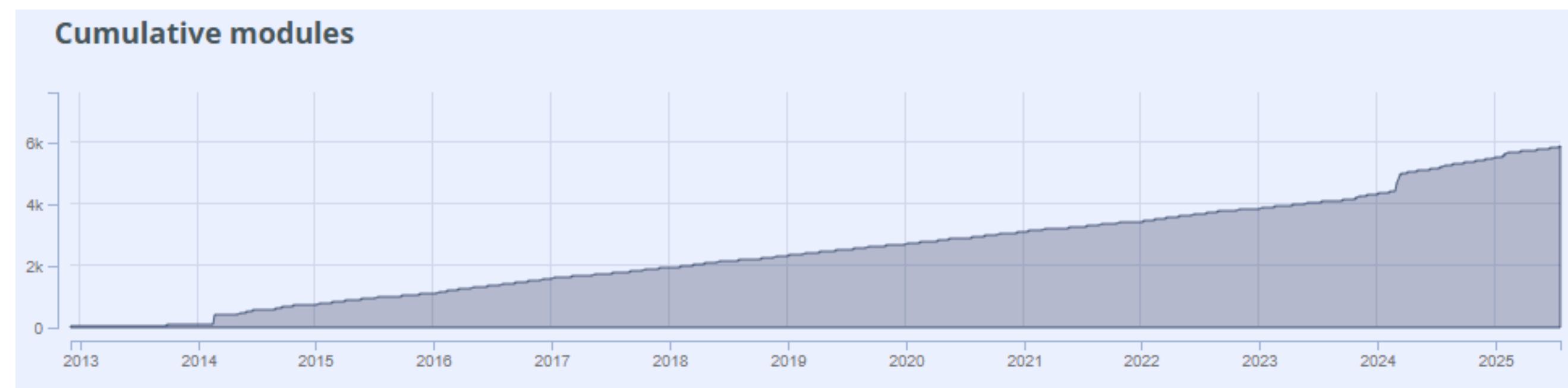


# Características de Lua



# LuaRocks

Gerenciador de  
pacotes de Lua



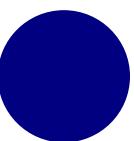
# **Como buildar, instalar e rodar**

No Linux:

Apenas: sudo apt install lua5.4 liblua5.4-dev



Para usar  
a API C



# **Como buildar, instalar e rodar**

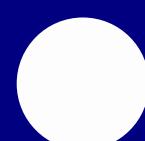
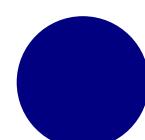
No Linux, por padrão, as headers e as bibliotecas serão armazenadas em:

**headers:**

`/usr/include/lua5.4`

**bibliotecas:**

`/usr/lib/x86_64-linux-gnu`

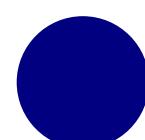


# **Como buildar, instalar e rodar**

No Windows:

Ou se instala apenas os binários de Lua,  
via <https://luabinaries.sourceforge.net>

A pasta dos binários  
deve estar igual a do  
meu repositório, pode  
baixar ela



# **Como buildar, instalar e rodar**

No Windows:

Ou se compila o código-fonte. Será  
necessário o compilador GCC via MinGW,  
preferencialmente o de 64 bits.

Material no README do  
repositório do Workshop  
sobre!



# Como buildar, instalar e rodar

No Windows:

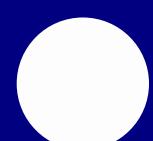
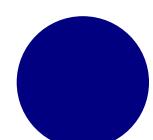
Com o **código-fonte** desempacotado, baixe um **script** (como o do meu GitHub) para compilar ele. Ao executá-lo, com a pasta do **código-fonte** na mesma pasta que o **script**, será retornada uma pasta **lua** com tudo necessário para utilizar a linguagem.



# Como buildar, instalar e rodar

No Windows:

Agora, tanto quem instalou pelos binários ou compilou, a **pasta** que contém o interpretador **lua.exe** e o compilador **luac.exe** deve ser adicionada como variável de ambiente.



# Como buildar, instalar e rodar

Para testar se tudo está funcionando:

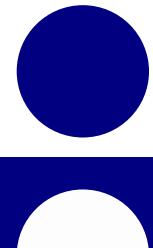
versão que  
você baixou

Abra o terminal, prompt de comando (cmd)  
ou o PowerShell e rode comando “**lua**”

→  
Lua 5.4.8 Copyright (C) 1994-2025 Lua.org, PUC-Rio

Se aparecer:

Tudo deu certo!



# Como buildar, instalar e rodar

Com esse comando, acabamos  
de entrar no REPL de Lua

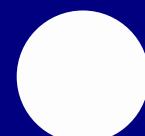
**Read-Evaluate-Print Loop**

Lê, avalia e imprime em loop

código

o resultado

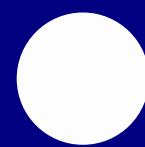
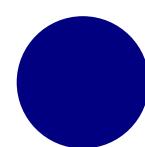
Também chamado de  
**Modo Interativo**



# Como buildar, instalar e rodar

Vamos instalar a extensão  
**'Code Runner'** no VS Code e  
configurá-la

→ Rodar no terminal



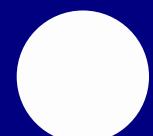
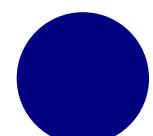
# Como buildar, instalar e rodar

Vamos instalar a extensão ‘Lua’  
no VS Code, do [sumneko](#)

- Lua Language Server

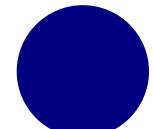


# **Programando em Lua**



# Programando em Lua

```
print("Olá, mundo!")
```



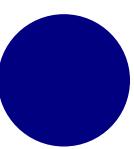
# Programando em Lua

-- Comentário

Comentários:

--[[

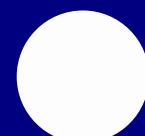
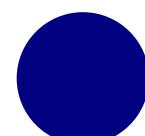
Comentário  
Multilinha



# Tipos

## number

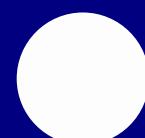
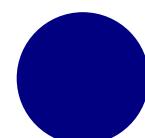
- Tanto float quanto int
- Float “por padrão”



# Tipos

## string

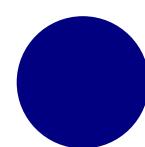
- Cadeias de caracteres
- Podem ser definidas com “ ” ou ‘ ’



# Tipos

## string

- Podem ter múltiplas linhas com [[  
]]
- São imutáveis



# Tipos

**string**

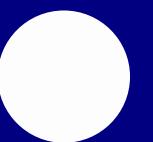
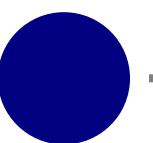
- São variáveis, não objetos, logo, não podem ser indexadas.

Mas existem alternativas.

**str[1]**



não  
funciona!

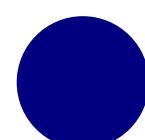


# Tipos

**boolean** →

Igual outras  
linguagens

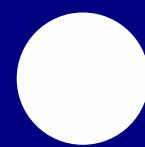
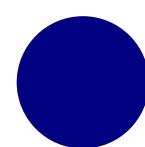
- 0 também é true
- String vazia também é true



# Tipos

nil → nulo

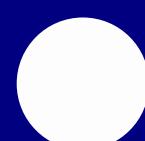
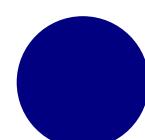
- Expressão retorna nil = falsa
- Atribuir nil → G.C apaga da memória



# Tipos

**function** → valores de  
1º classe

- Podem ser armazenadas em variáveis,  
passadas como argumentos e retornadas  
por outras funções



# Tipos userdata

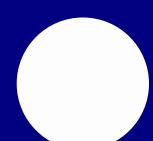
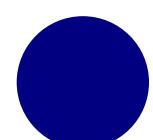
- API C
- Basicamente ponteiros  
para blocos de memória

Não se cria userdata  
em Lua diretamente,  
precisamos de  
outras fontes

# Tipos

## thread

- Corrotinas
- Adicionado na 5.0



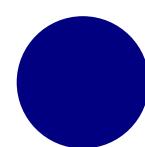
# Tipos

## Tabela

{ }

construtor

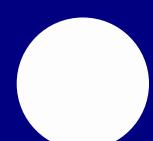
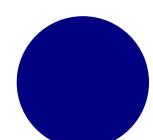
- **Arrays associativos:** podem ser tanto arrays, como dicionários ou os dois juntos
- **Metatabelas** extendem sua funcionalidade!



# Tipos

Passagem por **valor** vs. **referência**

number, string, boolean, nil,  
table, function, userdata, thread





# Passagem por valor



## Tipos

Ao atribuir o valor de uma variável a outra, o valor da variável original é **copiado** para a nova variável, **ocupando um novo endereço** de memória.





# Passagem por referência



## Tipos

Ao atribuir uma variável a outra, a nova variável apenas passa a **referenciar** o **mesmo endereço** de memória da variável original.





# Passagem por referência



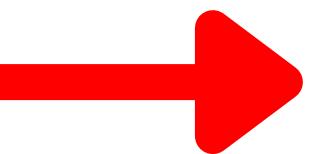
## Tipos

Se várias variáveis apontam para o mesmo endereço, para apagá-lo de verdade (**GC**), todas as variáveis com referências para esse endereço devem se tornar **nil**.

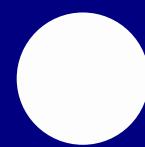
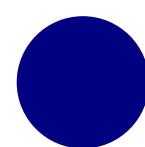


# Tipos

Função:  
**type()**



Informa o tipo  
de uma variável  
ou expressão



# Tipos

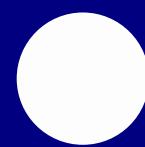
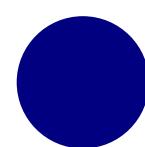
Funções:

**tonumber()**

**toString()**



Quando possível,  
convertem uma  
variável de um  
tipo para outro



# Operadores

Matemáticos:

+ - \* / % ^ //

→ Divisão de piso

5 / 2 = 2.5

5 // 2 = 2

Exponencial



# Operadores

Bitwise:

& ~ | << >>

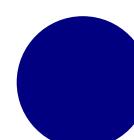
Igual C e Python

$8 = 1000$

$8 \& 7 = 0000$

$7 = 0111$

$8 | 7 = 1111$



# Operadores

Lógicos:

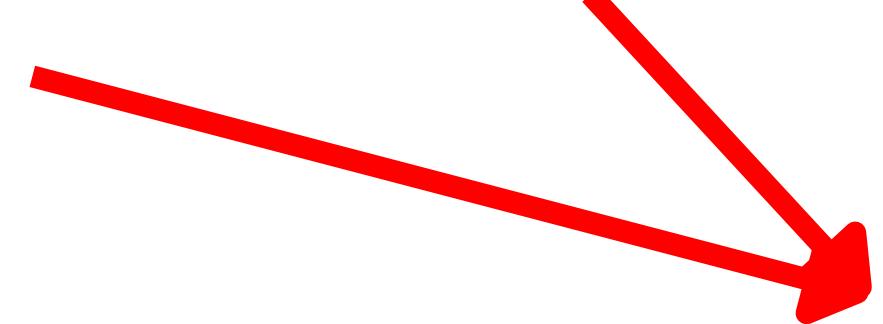
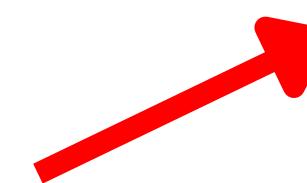
and

or

not

inverte o valor de uma  
expressão e retorna  
um valor booleano

o segundo operando  
só é avaliado se  
necessário



# Operadores

Lógicos:

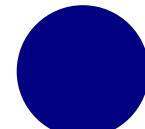
a and b

and

**1º operando false** = Retorna o **1º operando**

**1º operando true** = Retorna o **2º operando**

**Qualquer valor diferente de nil ou **false** é **true****



# Operadores

Lógicos:

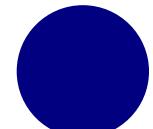
a or b

or

**1º operando false** = Retorna o **2º operando**

**1º operando true** = Retorna o **1º operando**

**Qualquer valor diferente de nil ou false é true**



# Operadores

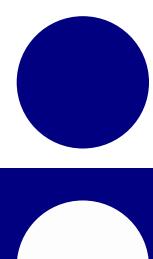
Lógicos:

```
function initx (v)
  x = v or 100
end
```

or



se **v** for nil ou  
**false**, **x** será 100



# Operadores

Lógicos:

not

Transforma qualquer  
valor em booleano e  
inverte seu valor



a = 15	→ "true"
not a	→ false
not not a	→ true

# Operadores

Operador de comprimento:

#



Retorna o comprimento  
de uma string ou tabela

Contanto que não  
tenha “buracos”.



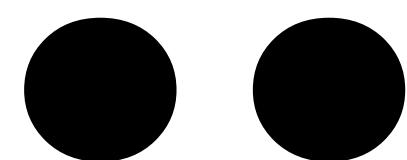
Apenas a parte da tabela  
utilizada como array.



# Operadores

Operador de comprimento:

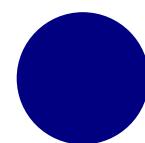
Caso um dos elementos  
não for string, quando  
possível, será  
transformado em uma.



**Concatena duas strings**

“João” .. “ da Silva”

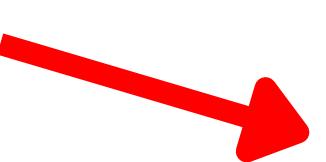
→ “João da Silva”



# Operadores

Relacionais:

< >    <= >=    ==    ~=



Equivalente à  
!= em outras  
linguagens

20 > 15 → true

# Estruturas de Controle

**if:**

**if** *condição* **then**

... código ...

**elseif** *condição* **then**

... código ...

**else**

... código ...

**end**



# Estruturas de Controle

**while:**

**break** para a  
sua execução

```
while condição do  
    ... código ...  
end
```

↑ Repete  
enquanto  
for **true**

# Estruturas de Controle

while:

```
local i = 0
while i < 10 do
    print("oi")
    i = i + 1
end
```

→ imprime "oi"  
10 vezes

# Estruturas de Controle

**repeat:**

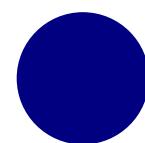
**break** para a  
**sua execução**

**repeat**

... código ...

**until** *condição*

→ Repete enquanto  
for **false**



# Estruturas de Controle

for numérico:

**for i = início, até, passo do**

break para a  
sua execução

**end**

... código ...

opcional



# Estruturas de Controle

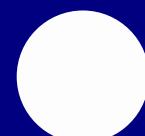
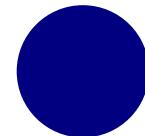
for numérico:

**for i = início, até, passo**

i = valor inicial da iteração

até = até onde a iteração vai

passo: comportamento da iteração (soma 1 por padrão)



# Estruturas de Controle

for numérico:

```
for i = 1, 5, 1 do
    print("Oi pela " .. i .. "º vez!")
end
```

imprime “Oi pela iº vez!”  
5 vezes

# Estruturas de Controle

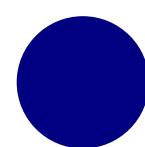
Também existe o

**for genérico**



**pairs()**  
**ipairs()**

Iremos ver depois que  
passarmos pelas tabelas



# Blocos e Escopo Léxico

**Bloco** = Algum trecho de código qualquer

Se usa a keyword **local** ao definir uma variável para especificar que essa variável pertence apenas à seu **bloco** e quaisquer blocos interiores à este.

local x = 15

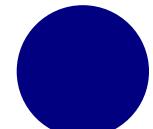
Até o nosso programa inteiro é um bloco!

Léxico = conjunto de palavras de uma linguagem

# Blocos e Escopo Léxico

Estruturas de controle (**if, while, repeat, for**)  
são exemplos de **blocos**.

Qualquer coisa que utilize **do** e **end** é um **bloco**.  
Podemos criar um **bloco simples** apenas com  
**do-end**. **Funções** também são **blocos**.

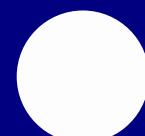
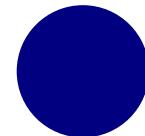


# Blocos e Escopo Léxico

Caso não usemos **local**, a variável é acessível em **todo** o programa. Ela será **global**.

x = 15

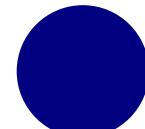
**Tudo** que é **global** em nosso programa fica na tabela **\_G**. Inclusive todas as bibliotecas.



# Blocos e Escopo Léxico

A função `print`, `type`, `tonumber`, `tostring`, as funções das bibliotecas `math`, `string`, `io`, `os` e etc também ficam na tabela `_G` (global).

É **boa prática** não “bagunçar” (encher) a tabela `_G` (global). Use `local` sempre que for possível.

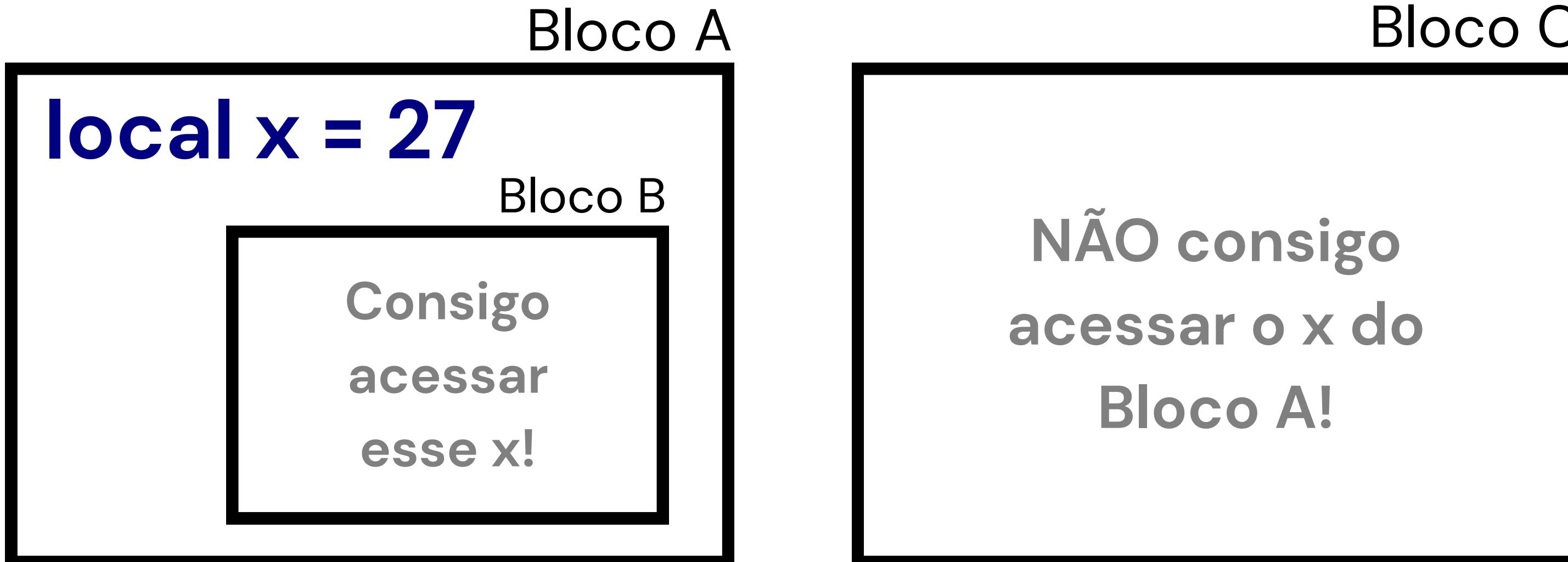


# Blocos e Escopo Léxico

Caso definamos um **bloco** dentro de **outro bloco**, o **bloco interior** **também** terá acesso às variáveis **locais** do seu **bloco exterior**.



# Blocos e Escopo Léxico



Bloco P

Pode ser o  
nosso  
programa

Consegui acessar **tudo!**

Bloco A

**local x = 27**

Bloco B

Tenho  
acesso  
à z!

Consegui  
acessar  
esse x!

Tenho  
acesso  
à z!

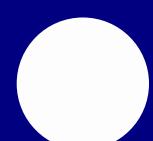
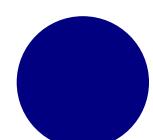
Bloco C

**z = 'string global!'**

NÃO consegui  
acessar o x do  
Bloco A!

# Blocos e Escopo Léxico

Existe uma tabela chamada `_ENV` que inicialmente aponta para (imita) `_G`. Essa tabela `_ENV` serve para definir um ambiente e pode ser útil principalmente para testes. Podemos mudar seu comportamento e alterar seu conteúdo, mas **não é recomendado**.

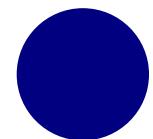


# Variáveis

Pode-se criar uma variável  
apenas escrevendo seu nome:

minha\_var

Se assim definida, ela será **global**  
e terá valor **nil**.



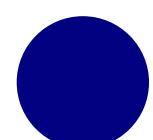
# Variáveis

local minha\_var

Se assim definida, ela será **local** e terá valor **nil**.

local minha\_var = 1337

Se assim definida, ela será **local**, e terá como valor o número **1337**.



# Variáveis

Sempre que não definirmos um valor para a variável ela terá valor **nil**.

Sempre que não definirmos uma variável como **local** ela será **global**.



# Variáveis

Podemos definir mais de uma variável ao mesmo tempo:

local x, y, z = 100, 200

Se utilizarmos **local**, todas as variáveis serão **locais**. Nesse caso, z será **nil**.

Caso não utilizemos, todas serão globais

# Variáveis

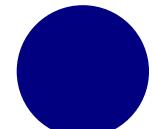
Declarando algumas variáveis:

local nome = "João"

local sobrenome = 'da Silva'

local idade = 21

local dinheiro\_na\_conta = 259.37



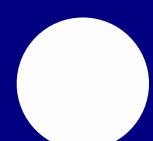
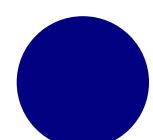
# Variáveis

Declarando algumas variáveis:

```
local nome_completo = nome .. " " .. sobrenome
```

```
local esta_cursando = false
```

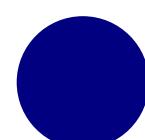
```
local nome_parceiro
```



# Tabelas

São a única estrutura  
de dados disponível  
por padrão em Lua

forma de  
organizar dados



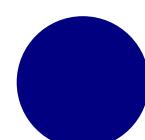
# Tabelas

Variáveis do tipo **table** guardam uma **referência** para uma tabela

Variáveis do tipo  
**table** guardam uma  
referência ao  
endereço de  
memória da tabela

```
local x = "sou_string"  
print(x) → "sou_string"
```

```
local tab = { 1, 2, 3, "abc", nome = "Gabriel"}  
print(tab) → table: 0000000000071a120
```



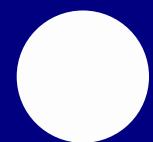
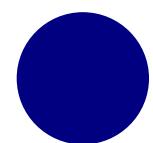
# Tabelas

Cria uma tabela  
nova, define seus  
elementos e já  
atribui ela a uma  
variável →

Para **inicializar** uma  
**nova** tabela utilizamos  
o construtor {}  
local minha\_tabela = { "oi", "tchau"}

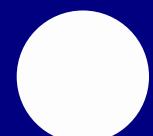
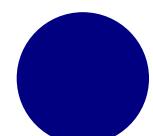
# Tabelas

Uma **tabela** pode  
guardar mais de um tipo  
de valor, seja **string**,  
**número** e até **função**.



# Tabelas

Podemos aumentar ou diminuir o tamanho das tabelas dinamicamente no meio do nosso código.



# Tabelas

Podem guardar dados como um array:

Inicializando uma tabela apenas como array:

```
local tabela_array = { 12, 48, 95, 30, 245 }
```

# Tabelas

Podem guardar dados como um array associativo (chave-valor):

Iniciando uma tabela apenas como array-associativo:

```
local tabela_assoc = { nome = "João", ID = 1337 }
```

hash-maps

# Tabelas

Podem guardar dados em uma  
tabela misturando os dois jeitos:

Inicializando uma tabela com parte array e chave–valor:

```
local tabelaMix = { "Casa", 1922, cidade = "Pelotas" }
```

# Tabelas

## Chaves/Índices e Valores

```
local tabela_assoc = { nome = "João", ID = 1337 }
```

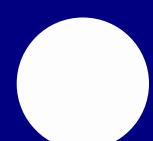
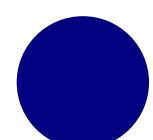
nome e ID são **chaves** → também chamadas  
de índice  
"João" e 1337 são os **valores** dessas **chaves**

# Tabelas

Tabelas são indexadas → por suas **chaves**

Indexar é o ato de acessar o valor de uma tabela por meio de uma chave específica.

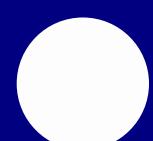
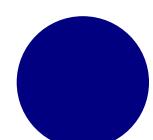
Na tabela exemplo **tabela\_assoc**, na chave “nome”, temos “João” como valor.



# Tabelas

**Chaves/Índices** em tabelas podem ser de qualquer valor!

Em uma tabela, os elementos em que definimos **explicitamente** a sua chave via **chave-valor**, geralmente usamos **strings** como **chaves**. Ao definir strings como chaves, não precisamos usar “ ”



# Tabelas

Nesse exemplo essa tabela só tem parte chave-valor mesmo, mas nem sempre é o caso

Esses elementos são da parte chave-valor da tabela!

array associativo

Definindo explicitamente as chaves

```
local tabela_assoc = { nome = "João", ID = 1337 }
```

# Tabelas

Em uma tabela, os elementos em que suas **chaves/índices** são **números** sequenciais (começando em 1), esses elementos serão da **parte array** da tabela!

array comum que  
conhecemos, elementos  
em sequência



# Tabelas

Nesse exemplo essa tabela só tem parte array mesmo, mas nem sempre é o caso

Esses elementos são da parte array da tabela!

NÃO estamos definindo  
explicitamente as chaves

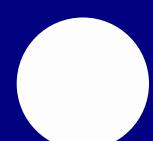
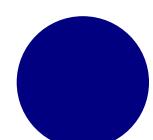


```
local tabela_nums = { 12, 20, 26, 34, 48, 56, 88 }
```

# Tabelas

```
local tabela_nums = { 12, 20, 26, 34, 48, 56, 88 }
```

Não definir as **chaves** explicitamente fará o elemento ser da **parte array** da **tabela**, logo, terá um **número sequencial** como **chave**

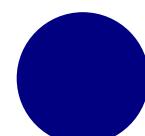


# Tabelas

## RESUMINDO:

Tabelas podem possuir uma **parte array**, uma parte **array associativo (chave–valor)** e **ambos ao mesmo tempo**.

A principal diferença dos dois é a sintaxe prática para defini-los. Em **chave–valor** explicitamos a chave, já em **array** não precisamos explicitar ao definir pois o índice já é um número sequencial.



1.

# Tabelas

## Outras diferenças:

A parte **chave–valor** de uma tabela não faz questão de manter alguma ordem, independente da forma como definirmos seus elementos.

```
local tabela_assoc = { nome = "João", ID = 1337 }      Não tem  
local tabela_assoc = { ID = 1337, nome = "João" }      diferença!
```

1.

```
local tabela_assoc = { nome = "João", ID = 1337 }
```

# Tabelas

## Outras diferenças:

A implicação disso é que a função pairs() irá retornar os pares chave-valor em uma ordem aleatória

```
for key, value in pairs(tabela_assoc) do  
    print(key, value)  
end
```

pode retornar  
dos dois jeitos

nome = "João"
ID = 1337
ID = 1337
nome = "João"



2.

# Tabelas

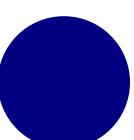
## Outras diferenças:

A parte **array** de uma tabela se importa,  
sim, com a ordem de seus elementos.

```
local tabela_nums = { 15, 20, 35 }
```

```
local tabela_nums = { 35, 15, 20 }
```

TEM  
diferença!



```
local tabela_nums = { 15, 20, 35 }
```

## Tabelas

2.

### Outras diferenças:

A implicação disso é que tanto a função **pairs()** quanto a **ipairs()** irão retornar a parte array de uma tabela em ordem.  
**ipairs()** serve apenas para parte array.

```
for key, value in pairs(tabela_nums) do  
    print(key, value)  
end
```

só pode retornar  
desse jeito

1	15
2	20
3	35



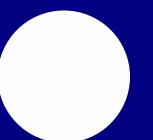
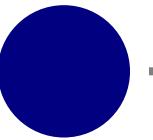
# Tabelas

## Outras diferenças:

O operador de tamanho (#) só irá retornar o tamanho da **parte array** de uma tabela.

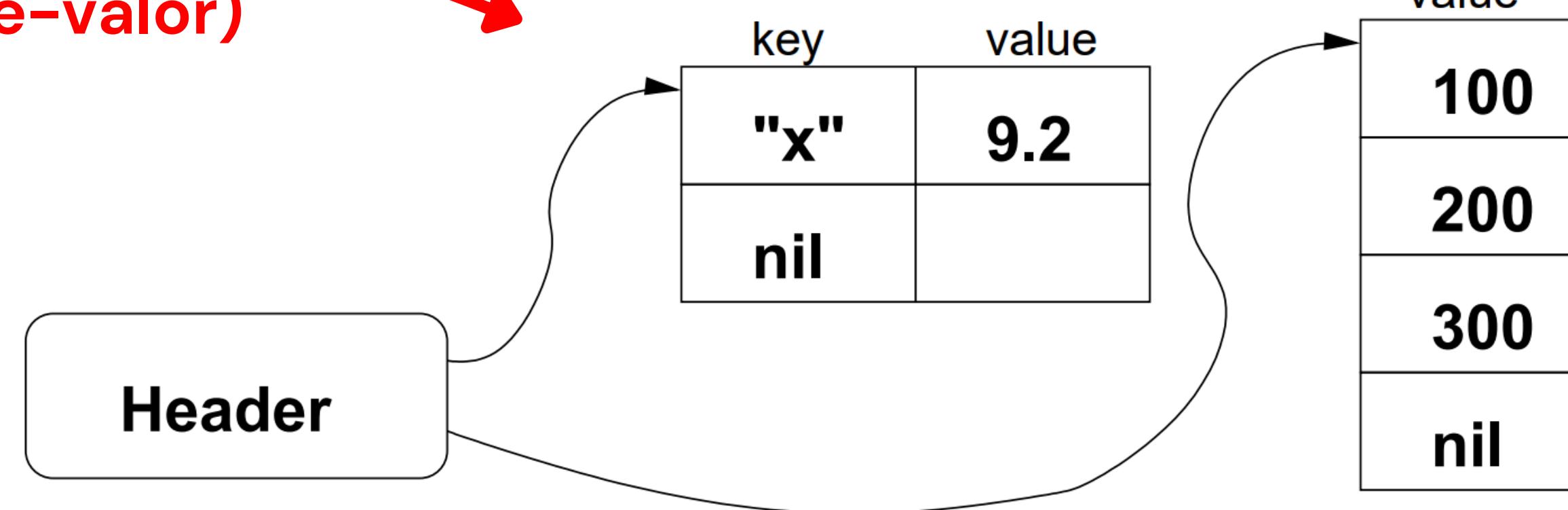
```
local tabela_mix = { nome = "João", ID = 1337, 15, 20, 35 }
```

**#tabela\_mix** → retorna 3, não 5



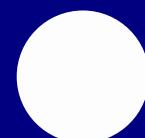
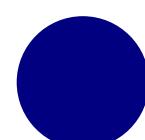
# Tabelas

parte array associativo  
(chave–valor)



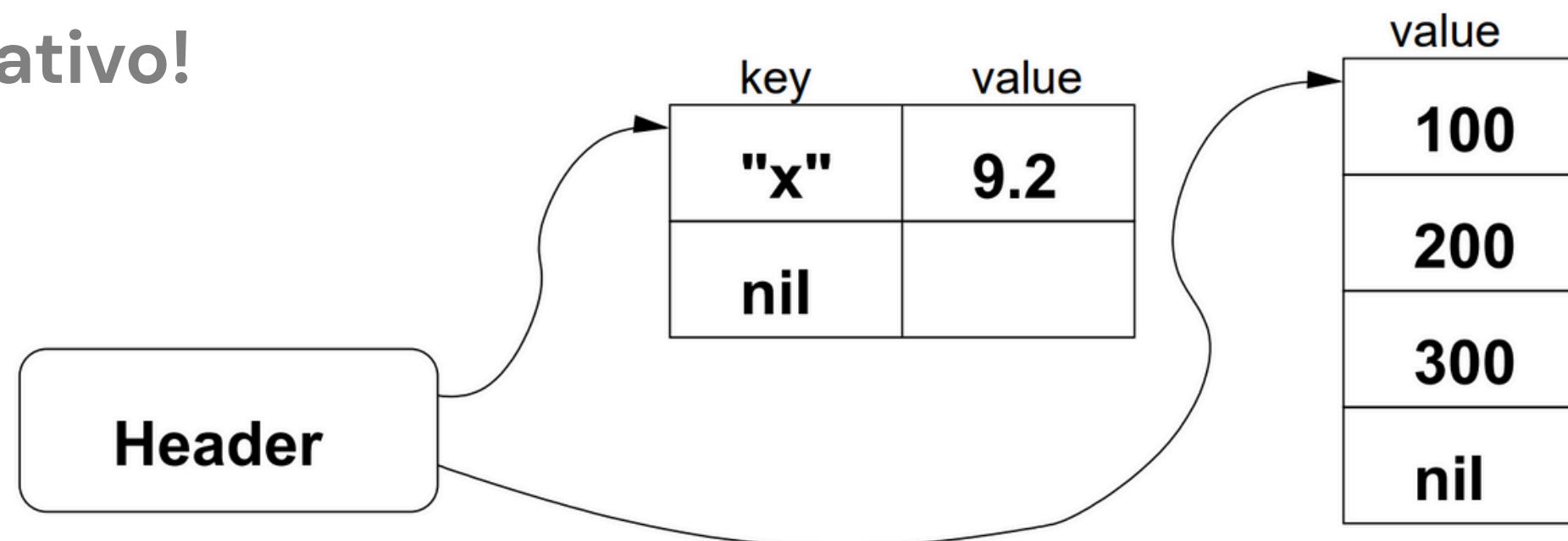
A linguagem sabe como lidar e interpretar como que a tabela está sendo utilizada por meio de um algoritmo, podendo ter tanto uma parte array como uma parte chave–valor ao mesmo tempo.

Acabam onde a chave é nil



Essa tabela possui tanto  
parte array como parte  
array associativo!

# Tabelas



```
local tabela_exemplo = { x = 9.2, 100, 200, 300 }
```

# Tabelas

```
local tabela_nums = { 12, 20, 26, 34, 48, 56, 88 }
```

**Nos arrays os índices começam em 1, não 0**

tabela\_nums[0] → retorna **nil**

→ sintaxe para acessar elemento  
do array é igual da linguagem C



# Tabelas

Até podemos adicionar o número **0** como chave na tabela, mas ele de forma nenhuma virá a ser da parte array da tabela, logo não será integrado com funções de Lua que atuam sobre a parte array de tabelas, como por exemplo **ipairs()**.

A decorative horizontal line consisting of a dashed line with two solid circular markers, one dark blue and one white, positioned at the ends.

# Tabelas

```
local tabela_nums = { 12, 20, 26, 34, 48, 56, 88 }
```

tabela\_nums[0] → retorna nil

tabela\_nums[1] → retorna 12

tabela\_nums[2] → retorna 20

tabela\_nums[3] → retorna 26 etc...

→ sintaxe para acessar  
elemento do array é  
igual da linguagem C

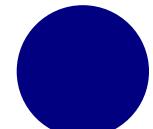
# Tabelas

Acessando elementos:

```
local tabela_nums = { 15, 20, 35 }
```

Para arrays, podemos usar a  
sintaxe semelhante à C:

tabela\_nums[1] → retorna 15



# Tabelas

## Acessando elementos:

```
local tabela_assoc = { nome = "João", ID = 1337 }
```

**Para chave-valor, ainda podemos usar a sintaxe semelhante à C, mas com uma diferença:**

tabela\_assoc["nome"] → retorna "João"

Para chaves que são strings, para acessar elementos precisamos usar " " na indexação.

# Tabelas

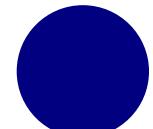
Acessando elementos:

```
local tabela_assoc = { nome = "João", ID = 1337 }
```

**Mas é mais comum utilizar essa sintaxe:**

tabela\_assoc.nome → retorna "João"

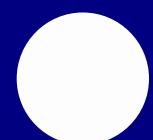
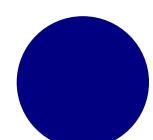
Açúcar sintático (syntax sugar)



# Tabelas

Podemos criar elementos novos  
depois de já ter inicializado a tabela:

É só acessar um **índice/chave**  
novo e atribuir um valor a ele.



# Tabelas

array

Criando novos elementos:

```
local tabela_nums = { 15, 20, 35 }
```

Se fizermos isso



```
tabela_nums[4] = 99
```

A tabela  
fica assim



```
local tabela_nums = { 15, 20, 35, 99 }
```

# Tabelas

chave–valor

Criando novos elementos:

```
local tabela_assoc= {}
```

Se fizermos isso



```
tabela_assoc.nome = "Felipe"
```

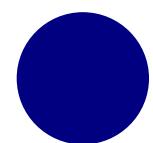
A tabela  
fica assim



```
local tabela_assoc = { nome = "Felipe" }
```

# Tabelas

Se acessarmos um **índice** que  
**não existe**, retornará valor **nil**,  
**não dará erro (por padrão)**!



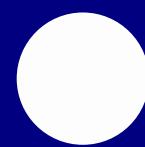
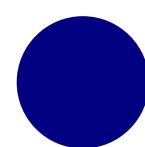
# Tabelas

array

Acessando elementos que não existem:

```
local tabela_nums = { 15, 20, 35 }
```

```
print(tabela_nums[4]) → retorna nil
```



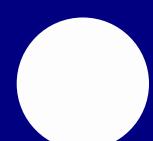
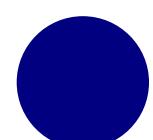
# Tabelas

chave–valor

Acessando elementos que não existem:

```
local tabela_assoc = { nome = "João"}
```

```
print(tabela_assoc.numero) → retorna nil
```

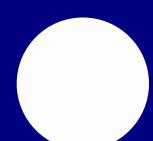
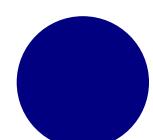


# Tabelas

Em **arrays**, cuidado ao criar  
elementos em índices fora da  
sequência de números de  
índice!



Vai criar  
**buracos nil** no  
**array** da tabela,  
quebrando seu  
funcionamento!



# Tabelas

```
local tabela_nums = { 12, 20, 26, 34, 48, 56, 88 }
```

```
tabela_nums[12] = "Qualquer coisa"
```

Como **arrays** de Lua são apenas sequências de números começando em 1 usados como índices/chaves e sem buracos no meio, **tabela\_nums[12]** não compõe a parte **array**, mas sim a parte **chave–valor**.

7 elementos

#tabela\_nums → 7

A função ipairs() não irá conseguir achar tabela\_nums[12], já pairs( ), sim.

# Tabelas

Adicionando elemento ao final do array (append):

```
local tabela_nums = { 15, 20, 35 }
```

Se fizermos isso

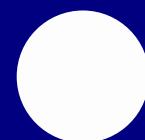
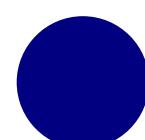


```
tabela_nums[#tabela_nums + 1] = 99
```

A tabela  
fica assim



```
local tabela_nums = { 15, 20, 35, 99 }
```



# Tabelas

Removendo elemento ao final do array (pop):

```
local tabela_nums = { 15, 20, 35 }
```

Se fizermos isso



```
tabela_nums[#tabela_nums] = nil
```

A tabela  
fica assim



```
local tabela_nums = { 15, 20 }
```



# Tabelas

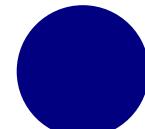
Podemos usar variáveis (o valor delas) para definir valores e usá-las como índice:

```
local x = 2
```

```
local tabela_nums = { 15, 20, 35 }
```

tabela\_nums[x] → retorna 20

tabela\_nums[4] = x → tabela fica { 15, 20, 35, 2 }



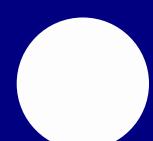
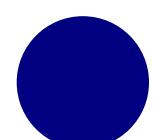
# Tabelas

Podemos guardar **funções** dentro de tabelas:

```
local tabelaComFuncao = { msg = print }
```

tabelaComFuncao.msg("oi") → imprime "oi"

Estamos atribuindo a função print à chave 'msg', agora podemos usar tabela.msg() como a função print.

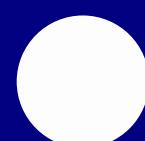
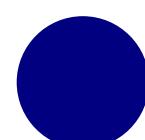


# Tabelas

Podemos guardar **tabelas** dentro de tabelas:

```
local tabelaComTabela = {  
    endereço = { Rua = "R. das Flores", CEP = "96010-610" }  
}
```

tabelaComTabela.endereço.Rua → imprime "R. das Flores"



# Tabelas

## Exercício Prático nº 1

Crie um **tabela como chave-valor** inicializando **duas strings** nela. Faça com que exista um terceiro elemento com chave '**Resposta**' que seja uma string com essas **duas strings** concatenadas. Imprima esta no terminal.

Acessar elemento:  
`tabela.chave`

Concatenação:  
`string1 .. string2`

# Tabelas

```
1 local strings = {string1 = "Eu fui", string2 = " concatenada."}  
2  
3 strings.Resposta = strings.string1 .. strings.string2  
4  
5 print(strings.Resposta)
```



# Estruturas de Controle

**for genérico:**

Serve para percorrer os valores

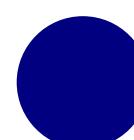
retornados por uma função iteradora:

**pairs()** → percorre todas as chaves de uma tabela

**ipairs()** → percorre os índices da parte array de uma tabela

**io.lines()** → percorre linhas de um arquivo

tanto a parte array  
como a parte  
chave-valor



# Estruturas de Controle

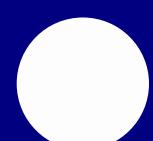
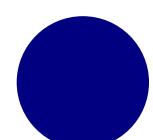
**pairs()**: → percorre todas as chaves de uma tabela

**for chave, valor in pairs(tabela)**

... código ...

**break** para a  
sua execução

**end**



# pairs():

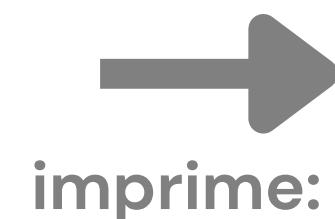
```
local coisas = { nome = "Gabriel", num = 7, 15, 20, 35 }
```

parte array  
começa aqui

```
for chave, valor in pairs(coisas)
```

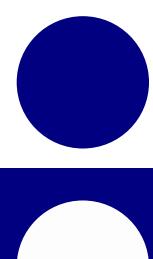
```
    print(chave, valor)
```

```
end
```



parte chave-valor, é iterada  
em ordem aleatória

1	15
2	20
3	35
nome	Gabriel
num	7



# Estruturas de Controle

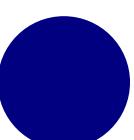
**ipairs()**: → percorre os índices de um array,  
indo de 1 até o n-ésimo elemento

**for índice, valor in ipairs(tabela)**

... código ...

**break** para a  
sua execução

**end**



# ipairs():

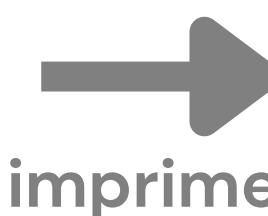
```
local nums = { 50, 150, 250, 350, 450 }
```

```
for chave, valor in pairs(nums)
    nums[chave] = valor + 50
    print(chave, chave[valor])
end
```

→ usar apenas 'valor' iria retornar os valores originais da tabela pegos pela função iteradora

Mesmo que a tabela tivesse parte chave-valor iria apenas ser iterado sobre a parte array apenas.

1	100
2	200
3	300
4	400
5	500



imprime:

# Estruturas de Controle

Lembrando que o **ipairs()** ao iterar sobre a **parte array** de uma tabela, só conseguirá chegar **até** o elemento anterior de onde tiver um **buraco (nil)** na tabela.

O ideal é que o **nil** seja sempre o final real do **array**, e não um **buraco** indesejado.



# Funções

Entrada → Saída  
ou simplesmente  
Executar algo

Funções são **blocos reutilizáveis** de código que você pode chamar a qualquer momento.

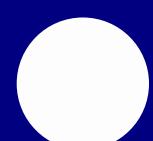
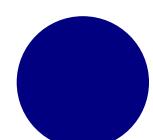
Elas **podem receber valores (chamados argumentos)** quando são chamadas, **fazem algo com ou sem eles** e **podem devolver um resultado**.

# Funções

**Parâmetros vs. Argumentos:**

**Argumentos** são os valores que você passa para os **parâmetros** da função quando chama ela.

**Parâmetros** são as variáveis da função que vão receber os valores passados como **argumentos**.  
Esses farão o trabalho pesado.



# Funções

Definição de função:

```
function nomeDaFuncao( parâmetros )
```

... código ...

```
return
```

Não  
esquecer! → **end**

Código pode operar  
sobre parâmetros ou não

Pode ter desde  
nenhum até vários



# Funções

Irá retornar um valor  
ou uma expressão  
computada dentro  
da função (ou nada)

→ **return**

Pode retornar mais de  
uma coisa usando ','

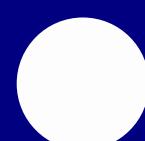
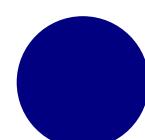


Pode ter desde  
nenhum até  
vários



Se a função chega em um  
return, ela retorna ele e para  
sua execução nele.

**"Seu trabalho estará pronto"**

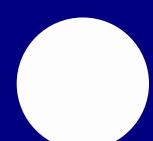
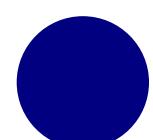


# Funções

Chamando uma função:

**nomeDaFuncao(argumentos)**

→ Executa a função



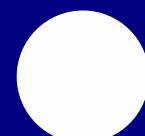
# Funções

Exemplo:

```
1 function soma(x, y)
2     return x + y
3 end
4
5 local num1 = 15
6 local num2 = 35
7
8 local result = soma(num1, num2)
```

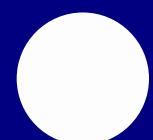
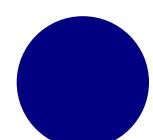
Por que soma está sublinhado?

result possui agora o  
valor 50



# Funções

Agora fica um pouco  
confuso, escute com  
atenção!



# Funções

Funções em Lua são na verdade valores de **primeira classe**, logo, podem ser guardados em **variáveis**!

```
function digaOi()
    print("Oi!")
end
```



na verdade é só um jeito diferente de escrever isso, que é o que está realmente acontecendo!

```
digaOi = function()
    print("Oi!")
end
```

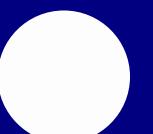
O jeito da esquerda de definir funções é apenas o açúcar sintático da linguagem!

Variável global `digaOi` guarda uma referência para essa função que executa esse código

# Funções

Todas as **funções** em Lua na verdade são apenas um bloco qualquer de código sem nome na forma **function(parâmetros) - código - end**.

Isso é o que podemos chamar de **função anônima**, todas as **funções** em **Lua** tecnicamente são assim. O que acontece é que podemos atribuir essas **funções** “que vivem soltas” à **variáveis** (**elas** que possuem nomes, não as funções), logo, a **variável** vai guardar uma **referência** dessa **função** e poderá agir como se fosse ela com () .



# Funções

**local imprimeMsg = print**



**imprimeMsg("oi")**

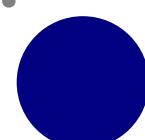


E sim, vai  
imprimir "oi".



imprimeMsg agora guarda uma  
referência da função print,  
podendo agora agir como ela.

Ao chamar uma **função**, a linguagem verifica se a **função** que foi chamada está definida e existe na **memória**. Como imprimeMsg aponta para o **mesmo endereço** que a função print aponta (no qual tem uma **função** que imprime os argumentos passados para o terminal), essa **função** é executada.



# Funções

Variáveis do tipo **function** guardam  
uma **referência** para uma função

Variáveis guardam  
uma referência ao  
endereço de  
memória da função



```
local x = 123  
print(x) → 123
```

```
local func = function() print("oi"); end  
print(func) → function: 000000000006ecfb0
```

# Funções

Cuidado ao atribuir  
funções à outras funções!

Exemplo:

```
1 local hora = function()
2     |     return "Data e hora: " .. os.date()
3 end
4
5 print(hora())
```

Define a função hora e imprime seu resultado.

( ) retornam o resultado!

```
7 local retornaHora = hora()
8 local funcaoHora = hora
```

Essas duas linhas NÃO fazem a mesma coisa!

A linha 7, **por usar ()**, atribui o **RETORNO** da função hora à variável retornaHora, logo, retornaHora, guardará, nesse exemplo, uma string.

Já a linha 8 atribui a **FUNÇÃO** hora **EM SI** à variável funcaoHora. No caso, uma referência para a função.

# Funções

Funções serem valores de primeira classe implica que podemos passar

funções como argumentos para outras funções!

Exemplo:

```
1 local fazAlgoComString = function(f, string)
2   |   f(string)
3 end
4
5 fazAlgoComString(print, "coisa louca né!")
```

Como passamos **print** como **argumento** para **f**, o **parâmetro** dentro da **função, f** irá agir como **print**.

# Funções

Podemos passar funções  
como **argumentos** para  
**outras funções!**

Outro exemplo:

```
1 local executar = function(f)
2     f()
3 end
4
5 executar(function() os.exit() end)
```

Dessa vez, **definindo a função a ser  
utilizada como argumento na própria  
chamada da função.**  
E sim, essa daí vai finalizar o programa.

Cuidado pra não  
se confundir!

# Funções

## Exemplo:

- 1 - a função **hello** é criada
- 2 - a função f é **chamada** usando a função **hello** como parâmetro g
- 3 - **hello** é **executada** como g em f
- 4 - variável 'x' de **g (hello)** retorna

```
1 local function f(g)
2     | return g()
3 end
4
5 local function hello()
6     | print("Olá!!!")
7     | local x = 10
8     | return x
9 end
10
11 local oi = f(hello)
12 print(oi)
```

# Funções

Funções podem retornar outras funções!

Exemplo:

```
1 local quero = function(string)
2     if string == "somar" then
3         return function(x, y)
4             return x + y
5     end
6     elseif string == "multiplicar" then
7         return function(x,y)
8             return x * y
9     end
10    end
11 end
12
13 local minhaFunc = quero("somar")
14 print(minhaFunc(1, 9)) --> retorna 10
```

# Funções e Closures

Agora que descobrimos que podemos retornar **funções** de **outras funções**, surge uma pergunta:

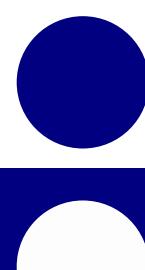
Se Lua possui **escopo léxico** orientado à blocos, e **funções** são blocos, o que acontece se eu quiser retornar uma **função interna** dentro da minha **função** e essa **função interna** tiver **valores locais** associados à ela? Eles vão sumir já que saíram do **escopo**?



# Funções e Closures

A resposta é não! E o nome disso são closures, um conceito da programação que é implementado de forma diferente dependendo da linguagem:

Em **Lua**, todos os **valores locais** à **função interna** a ser retornada irão ser armazenados como **upvalues**, uma categoria especial de variável no **escopo léxico**. Enquanto a **função que foi retornada** existir, **eles** existem. **Eles** permite que **funções** em **Lua** "lembrem" de variáveis mesmo depois que a **função original** já terminou.



# Funções e Closures

Exemplo  
clássico:

```
1 local criarContador = function()
2     local count = 0 --> isso é um upvalue para a função retornada
3     return function()
4         count = count + 1
5         return count
6     end
7 end
8
9 local c = criarContador()
10 print(c()) --> 1
11 print(c()) --> 2
12 print(c()) --> 3
```

# Funções

**Número variável de parâmetros (...):**

Podemos passar um **número variável** de **parâmetros** para uma **função** em sua **definição** com:

`function(...)`



... deve ser o último  
parâmetro sempre

# Funções

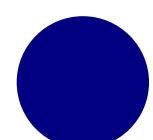
Iremos **iterar** sobre os vários **parâmetros** com a função iteradora **ipairs()**.

**nvalor** representa o número do valor que a iteração está

**valor** representa o valor em si

```
1 local calculaMedia = function(...)  
2     local numerador, denominador = 0, 0  
3  
4     for nvalor, valor in ipairs(...) do  
5         denominador = denominador + 1  
6         numerador = numerador + valor  
7     end  
8  
9     return numerador / denominador  
10    end  
11  
12    print(calculaMedia(5, 6, 7, 8, 9, 10))
```

retorna 7.5



# Funções e Tabelas

Assim como **tabelas**  
podem armazenar **funções**,  
as **funções** podem operar  
sobre **tabelas**



# Funções e Tabelas

## Tabelas com funções:

Chamamos as **funções** pelas suas  
**chaves** na **tabela**!

```
1 local imprimirNomes = {  
2     printMaria = function() print("Maria"); end,  
3     printGabriel = function() print("Gabriel"); end  
4 }  
5  
6 imprimirNomes.printMaria() --> imprime Maria
```

```
8     local maiorNum = function(x,y)  
9         if x > y then  
10             return x  
11         else  
12             return y  
13         end  
14     end  
15  
16     local minhasFuncs = {max = maiorNum}  
17  
18     print(minhasFuncs.max(30, 90))  
19     --> nesse caso imprimirá 90
```

# Funções e Tabelas

Funções  
operando sobre  
tabelas:

```
1 local amigos = {"Julia", "Matheus", "Leonel", "Olga"}  
2 local suasIdades = {15, 12, 40, 65}  
3  
4 local infoPessoas = function(nomes, idades)  
5     for i = 1, #nomes do  
6         print("-----")  
7         print("Nome: " .. nomes[i])  
8         print("Idade: " .. idades[i])  
9     end  
10    end  
11  
12    infoPessoas(amigos, suasIdades)
```

# Funções

## Exercício Prático nº 2

Crie uma **função** que receba como **parâmetro** uma **tabela** utilizada como array com números dentro.

A **função** deve retornar a soma desses números.

Crie uma **tabela** real para testar a sua **função**.

Acessar elemento:  
tabela[índice]

```
for i, v in ipairs(t) do  
    .. código ..  
end
```

# Funções

```
1 local soma = function(tabela)
2     local soma = 0
3     for _, valor in ipairs(tabela) do
4         soma = soma + valor
5     end
6
7     return soma
8 end
9
10 local numeros = {6, 7, 12, 25, 37, 48, 51}
11
12 print(soma(numeros))
```



# Bibliotecas de Lua

Agora que já vimos **escopo léxico, tabelas** e **funções**, tente rodar esse código:

```
1  for chave, valor in pairs(_G) do
2      print(string.format("%-15s= %s", chave, valor))
3  end
```

# Bibliotecas de Lua

Agora teste esse código com uma  
pequena mudança!



```
1 for chave, valor in pairs(_G.table) do
2     print(string.format("%-15s= %s", chave, valor))
3 end
```

# Bibliotecas de Lua - io

Tem funções que permitem interagir com o usuário via terminal, escrever e ler de arquivos e correlatos.

io.close  
io.flush  
io.input  
io.lines  
io.open  
io.output  
io.popen  
io.read  
io.stderr  
io.stdin  
io.stdout  
io.tmpfile  
io.type  
io.write

→ abre um arquivo para mexer nele

→ lê algo escrito do terminal

→ escreve texto para o terminal

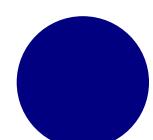
# Bibliotecas de Lua - io

## io.open

```
local file = io.open("arquivo.txt", "r")
```

Nesse exemplo, abre o “arquivo.txt” no modo “r” (read) (escrita) e retorna um identificador para esse arquivo na memória, permitindo agora operar sobre este.

Como a biblioteca `io.open` é escrita em C e `io.open` retorna um identificador para arquivos na memória, o retorno de `io.open` será do tipo **userdata**.



# Bibliotecas de Lua - io

## Entrada e saída

`io.write("O que quiser")` →

Extremamente similar à `print()`, mas como `print` usa `\n` no final, não é ideal para **ler input** na mesma linha! Então usamos `io.write("string")`

`local inputUser = io.read()` →

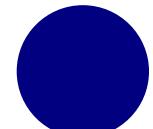
Irá aguardar **input** do usuário digitar algo no terminal, assim que ele digitar algo e dar **Enter**, irá capturar o que o usuário digitou e armazenará esse **input** na **variável** especificada no programa.

# Bibliotecas de Lua - io

## Entrada e saída

Exemplo:

```
1 io.write("Qual seu número favorito?: ")
2 local numFav = io.read()
3 print("Seu número favorito é: " .. numFav)
```



# Bibliotecas de Lua - io

## Exercício Prático nº 3

Pedir input:  
io.write("string")

Ler input:  
io.read()

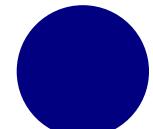
Para calcular a **área de um retângulo**, peça input do usuário quanto à **base** e **altura** do retângulo, assim como uma **unidade de medida (estética)** utilizando as **funções** io.write e io.read.

Calcule a **área desse retângulo** e a imprima da seguinte forma:

```
print("A área do retângulo é: " .. areaDoRetangulo .. unidadeDeMedida .. "^2")
```

# Bibliotecas de Lua - io

```
1 local calculaArea = function(base, altura)
2     return base * altura
3 end
4
5 io.write("Qual vai ser a base?: ")
6 local base = io.read()
7
8 io.write("Qual vai ser a altura?: ")
9 local altura = io.read()
10
11 io.write("Qual vai ser a unidade de medida?: ")
12 local unidadeDeMedida = io.read()
13
14 local areaDoRetangulo = calculaArea(base, altura)
15 print("A área do retângulo é: " .. areaDoRetangulo .. unidadeDeMedida .. "^2")
```



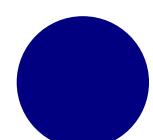
# Bibliotecas de Lua - math

Tem **valores e funções** que servem para lidar com **números**, **trigonometria**, **operações matemáticas** e mais.

`math.abs`  
`math.acos`  
`math.asin`  
`math.atan`  
`math.ceil`  
`math.cos`  
`math.deg`  
`math.exp`  
`math.floor`  
`math.fmod`  
`math.huge`  
`math.log`  
`math.max`  
`math.maxinteger`  
`math.min`  
`math.mininteger`  
`math.modf`  
`math.pi`  
`math.rad`  
`math.random`  
`math.randomseed`  
`math.sin`  
`math.sqrt`  
`math.tan`  
`math.tointeger`  
`math.type`  
`math.ult`

# Bibliotecas de Lua - math

```
1 print( math.pi )           --> não é função! é só o número de pi 3.141592...
2 print( math.sqrt(9) )       --> calcula raíz quadrada (nesse caso é 3)
3 print( math.abs(-15) )      --> valor absoluto (distância de 0) (nesse caso é 15)
4 print( math.random() )      --> um número aleatório entre 0-1 (depende da seed passada)
5 print( math.huge )          --> maior valor numérico da linguagem, serve como infinito
6 print( math.max(3, 9, 15, 20)) --> retorna o argumento de maior valor (nesse caso 20)
7 print( math.min(3, 9, 15, 20)) --> retorna o argumento de menor valor (nesse caso 3)
8 print( math.log(8, 2))       --> retorna o log do 1º argumento com o 2º como base (nesse caso 3)
9 print( math.type(3.14) )     --> retorna o tipo do argumento: "integer", "float" ou false caso falhe.
10 print( math.cos(math.pi))    --> retorna em radianos o cosseno do que lhe for passado como argumento (nesse caso -1)
11 print( math.floor(4.6) )      --> retorna apenas o valor inteiro do argumento passado (nesse caso 4)
12 print( math.ceil(4.2) )       --> retorna apenas o valor inteiro +1 do argumento passado (nesse caso 5)
13 print( math.modf(6.5) )      --> retorna o valor inteiro e o decimal do argumento passado (nesse caso 6 e 0.5)
```

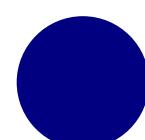


# Bibliotecas de Lua - math

## Exercício Prático nº 4

`math.modf(num)`  
→ inteiro, decimal

Leia o **input do usuário** assumindo que ele irá escrever um número não-inteiro e positivo qualquer e faça uma **função round()** que calcule esse número arredondado para cima ou para baixo baseado no valor decimal deste. Utilize a **função math.modf()** para isso. Imprima o resultado no terminal.



# Bibliotecas de Lua - math

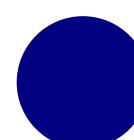
```
1 local round = function(numero)
2     local inteiro, decimal = math.modf(numero)
3
4     if decimal == 0 then print("Esse numero é inteiro!") end
5
6     if decimal >= 0.5 then
7         return inteiro + 1
8     else
9         return inteiro
10    end
11
12 end
13
14 io.write("Que número você quer arredondar?: ")
15 local numero = io.read()
16
17 local numeroArredondado = round(numero)
18
19 print(numeroArredondado)
```

# Bibliotecas de Lua - string

Tem **funções** que servem para lidar com e operar sobre **strings** e fazer **pattern matching** (correspondência de padrões).

`string.byte`  
`string.char`  
`string.dump`  
`string.find`  
`string.format`  
`string.gmatch`  
`string.gsub`  
`string.len`  
`string.lower`

`string.match`  
`string.pack`  
`string.packsize`  
`string.rep`  
`string.reverse`  
`string.sub`  
`string.unpack`  
`string.upper`



# Bibliotecas de Lua - string

string.byte  
string.char  
string.dump  
string.find  
string.format  
string.gmatch  
string.gsub  
string.len  
string.lower

string.match  
string.pack  
string.packsize  
string.rep  
string.reverse  
string.sub  
string.unpack  
string.upper

Podem operar sobre uma **string** diretamente ou com uma **variável** que contenha uma **string** guardada nela.

string.len("minha string")  
**ou**

local varstring = "minha string"  
string.len(varstring)

funciona do  
mesmo jeito!

# Bibliotecas de Lua - string

```
1 print( string.len("Onças e Leões") )           --> Retorna o número de bytes/caracteres da string passada.  
2                                         --> (inclusive Unicode contam como 2)  
3 print( string.reverse("Ao contrário!") )       --> Retorna a string passada de trás pra frente.  
4 print( string.format )                         --> função para formatar strings como o printf em C  
5                                         --> (usada bem com a função print em Lua)  
6 print( string.byte('ABC', 1) )                 --> Retorna o valores ASCII do caractere no índice  
7                                         --> especificado da string passada (nesse caso 65)  
8 print( string.char(65) )                       --> Retorna o caractere que possui esse valor ASCII (nesse caso 'A')  
9 print( string.upper("quero gritar!") )          --> Retorna uma string com todas as letras da string passada em maiúsculo  
10 print( string.lower("FALA BAIXO...") )         --> Retorna uma string com todas as letras da string passada em minúsculo  
11 print( string.rep("PET", 5, "repete") )        --> Retorna uma string repetindo o número de vezes dado pelo 2º arg.  
12                                         --> e separado pela string do 3º arg.
```

# Bibliotecas de Lua - string

```
14 print( string.find('Anglo', 'glo', 1))      --> 1º argumento = string original, 2º argumento = substring  
15                                         --> a ser procurada na string original. 3º argumento = em qual  
16                                         --> índice começar a procurar. retorna o índice onde começa a  
17                                         --> substring na string original e o índice onde a substring  
18                                         --> acaba. retorna false caso não ache. (nesse caso 3 e 5)  
19  
20 print( string.match("Anglo", "glo", 1) )    --> faz a mesma coisa que string.find, mas só retorna a substring  
21                                         --> caso ela existir na string original
```

# Bibliotecas de Lua - string

# Bibliotecas de Lua - string

**string.gmatch usa uma função iteradora sobre as substrings!**

```
1 local nome = "Tenho muitos amigos, e esses amigos me acompanham na vida."
2 local ocorrencias_da_substring = 0
3
4 for substring in string.gmatch(nome, "amigos") do    --> captura todas as instâncias da substring
5     ocorrencias_da_substring = ocorrencias_da_substring + 1
6     print(substring)
7 end
8
9 print(ocorrencias_da_substring)
```

# Bibliotecas de Lua - string

Pode também pode utilizar classe de caracteres para fazer correspondência de padrões!

```
12  
13 local texto = "Vamos brincar de esconde-esconde no bosque?"  
14  
15 for palavra in string.gmatch(texto, "%a+") do --> captura todas as palavras  
16     print(palavra)  
17 end  
18
```

# Bibliotecas de Lua - string

Pode também pode utilizar classe de caracteres para fazer correspondência de padrões!

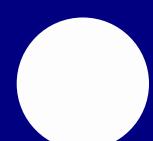
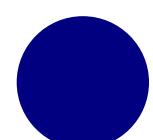
```
21 local idades = "Idades: 15, 42 e 7"  
22  
23 for num in string.gmatch(idades, "%d+") do      --> captura todos os números  
24     print(num)  
25 end
```

# Bibliotecas de Lua - string

Existe uma forma mais curta, um açúcar sintático (syntax sugar) para utilizar funções da biblioteca **string**:

Utilizando a `string.reverse` de exemplo, para chamar ela normalmente utilizamos: `string.reverse(suaString)`

Mas também podemos utilizar da seguinte forma:  
`suastring:reverse()` → o ":" irá jogar automaticamente a variável a sua esquerda como primeiro parâmetro da função! E isso serve para todas as funções da biblioteca string!



# Bibliotecas de Lua - os

Possui funções que utilizam de mecanismos do sistema operacional

os.clock
os.date
os.difftime
os.time
os.remove
os.rename
os.tmpname
os.setlocale
os.execute
os.exit
os.getenv

→ **Tempo**

→ **Arquivos**

→ **Execução ou o  
programa em si**

Sistema  
Operacional

# Bibliotecas de Lua - os

Tente fazer:  
`os.execute("python")`

```
1 print( os.time() )           --> retorna número de segundos desde a epoch
2 print( os.clock() )          --> retorna número de segundos desde o começo da execução do programa
3 print( os.date() )           --> retorna a data e a hora formatados bonitinhos
4 print( os.difftime(t1, t2) ) --> retorna diferença t1 - t2, onde esses são valores de os.time()

5

6 os.remove("arquivo")         --> remove esse arquivo e retorna true ou false se deu certo ou não
7 os.rename("arquivo", "nome")  --> renomeia o arquivo do nome antigo pro novo, retorno true ou false
8 print( os.tmpname() )        --> retorna com uma string do nome de um arquivo de uso temporário

9

10 os.setlocale(nil)           --> serve para alterar ou consultar as configurações regionais (locale) do ambiente.
11 os.execute()                --> O que for passado como argumento será rodado como comando no terminal.
12 os.exit()                   --> Fecha o programa.
13 print( os.getenv("USERNAME") ) --> Retorna o caminho para uma variável de ambiente ou false caso não dê.
```

# Bibliotecas de Lua - table

Possui **funções**  
para operar sobre e  
com **tabelas**.

`table.concat`  
`table.insert`  
`table.move`  
`table.pack`  
`table.remove`  
`table.sort`  
`table.unpack`



# Bibliotecas de Lua - table

**table.insert(tabela, índice, valor)**

**Serve apenas para a parte array**

Se o índice especificado não estiver no final, irá mover os outros índices para abrir espaço para o novo elemento

Opcional

Se não for especificado, insere o elemento no final+1 da tabela

# Bibliotecas de Lua - table

**table.insert(tabela, índice, valor)**

Exemplo:

```
1 local minhaT = {10, 30, 40}
2
3 table.insert(minhaT, 2, 20)
4
5 for indice, valor in ipairs(minhaT) do
6     print(indice, valor)
7 end
```

# Bibliotecas de Lua - table

**table.remove(tabela, índice)**

**Serve apenas para a parte array**

Se o índice especificado não estiver no final, irá mover os outros índices para cobrir o espaço que ficou sobrando

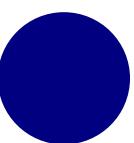
Opcional

Se não for especificado, remove o elemento no final da tabela

# Bibliotecas de Lua - table

```
local var = table.remove(tabela, índice)
```

O item removido também pode  
ser armazenado em uma variável



# Bibliotecas de Lua - table

**table.sort(tabela, função)**

Serve apenas para a parte array

Funciona também com strings,  
ordenando elas em ordem  
alfabética

Opcional

a > b para ordem  
decrescente

```
function(a, b)  
    return a < b  
end
```

Caso não seja  
especificada, irá  
ordenar a tabela  
do maior pro  
menor elemento

# Bibliotecas de Lua - table

## table.sort(tabela, função)

Exemplo:

```
1 local minhaT = {1, 4, 9, 2, 3, 5, 8}
2
3 table.sort(minhaT, function(a, b) return a < b; end)
4 --[[
5 nem era necessário adicionar essa função como argumento,
6 já que ela realiza o comportamento padrão de quando nem
7 é adicionado uma função, mas assim fica fácil de entender :)
8 ]]
9 for indice, valor in ipairs(minhaT) do
10   print(indice, valor)
11 end
```

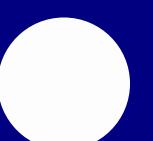
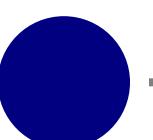
# Bibliotecas de Lua - table

```
table.concat(tabela, "separador")
```

Serve apenas para a parte array

↑  
Opcional

Retorna uma string com todos  
seus elementos concatenados  
em uma string só



# Bibliotecas de Lua - table

**table.concat(tabela, "separador")**

```
1 local fraseSep = {"Esse", "é", "o", "Minicurso", "de", "Lua", "!"}
2
3 local fraseJunta = table.concat(fraseSep, " ")
4
5 print(fraseJunta)
```

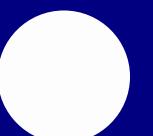
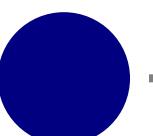
# Bibliotecas de Lua - table

`table.move(t1, t1[começo], t1[final], t2[começo], t2)`

**Serve apenas para a parte array**

Move o conteúdo da tabela **t1** de **t1[começo]** até **t1[final]** para a  
**tabela t2** em **t2[começo]**.

A **t2** pode ser a mesma tabela da **t1**, ou seja, tudo a mesma tabela.



# Bibliotecas de Lua - table

**table.move(t1, t1[começo], t1[final], t2[começo], t2)**

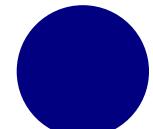
```
1 local primeira = {10, 20, 30, 40, 50}
2
3 local segunda = {}
4
5 table.move(primeira, 1, #primeira, 1, segunda)
6
7 for indice, valor in ipairs(segunda) do
8     print(indice, valor)
9 end
```

# Bibliotecas de Lua - table

```
local tab = table.pack(... elementos ...)
```

**Serve apenas para a parte array**

Realmente só guarda os  
elementos passados em ordem  
na a tabela nova especificada.



# Bibliotecas de Lua - table

**variáveis = table.unpack(tabela, começo, fim)**

**Serve apenas para a parte array**



Realmente desempacota a tabela, retornando os elementos dela em ordem para a quantidade de **variáveis** especificadas na ordem.



# Bibliotecas de Lua - table

**variáveis = table.unpack(tabela, começo, fim)**

```
1 local numerosSorteadosMegasena = {10, 2, 40, 46, 49, 58}
2
3 local a, b, c, d, e, f = table.unpack(numerosSorteadosMegasena)
4
5 local euQuero = table.pack(a, b, c, d, e, f)
6
7 for indice, valor in ipairs(euQuero) do
8     print(indice, valor)
9 end
```

# dofile

Executa todo o código de outro arquivo .lua.

É quase como se esse outro código fosse escrito dentro do código atual. É preciso passar o arquivo do outro código como string.

`dofile("outroarquivo.lua")`

precisa  
do .lua

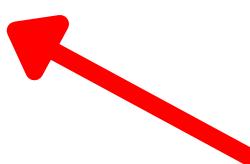
# dofile

arquivoPrincipal.lua

```
1 dofile("outroArquivo.lua")  
2  
3
```

outroArquivo.lua

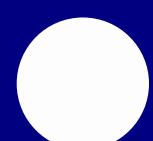
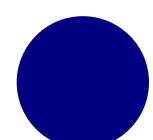
```
1 print("oi! eu estou no outro arquivo!")  
2  
3  
4
```



**por causa do dofile  
irá executar o print!**

# dofile

**dofile** não passa as variáveis **locais** do outro arquivo pro arquivo onde ele é chamado, **dofile** apenas passará as **variáveis globais!**



# Módulos

São basicamente jeitos de organizarmos e reutilizarmos código em arquivos lua diferentes. Existem **algumas** formas de utilizá-los. Algumas são más práticas, mas vamos elucidar algumas das boas práticas.



# Módulos

Em outro arquivo .lua criamos uma **tabela local** e no final **retornaremos ela**. Nessa **tabela adicionaremos as funções, tabelas e valores** que queremos passar pro nosso arquivo principal.

No arquivo principal criaremos uma variável e atribuiremos seu valor como o retorno da função require("nome\_do\_modulo")

No arquivo principal, para usar o módulo, usaremos a variável que criamos para ser a referência ao módulo e usaremos ela com a notação de tabela para referenciar os elementos do módulo.

NÃO precisa  
do .lua

# Módulos

## Arquivo principal

```
1 local modulo = require("ModuloUm")
2
3 modulo.mensagemLinda()
4 modulo.mensagemTriste()
5 print(modulo.meuNome)
```

funciona pois tudo  
isso está no módulo

## Módulo - ModuloUm

```
1 local Modulo = ...
2     mensagemLinda = function()
3         print("Lorem ipsum")
4     end,
5
6     mensagemTriste = function()
7         print("dolor sit")
8     end,
9
10    meuNome = "Julia"
11
12
13 return Modulo
```

# goto

Literalmente significa: “Vai pra essa parte do código”, sem tirar nem por.

É **controverso** e seu uso é **contraindicado** por poder complicar o código, mas ainda sim é uma **ferramenta útil as vezes**.

Pode emular a funcionalidade de um **continue**, que não existe em **Lua**.

Seu uso se dá por:  
**goto parteDoCodigo**  
**::parteDoCodigo::**

```
1 for i = 1, 10, 1 do
2     if i == 5 then
3         goto continue
4     end
5
6     print(i)
7
8 ::continue::
9 end
```

# assert

Testa uma expressão. Se ela for **false**,  
retorna a mensagem de erro que você  
quiser e encerra o programa.

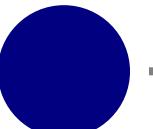
```
1 local idade = 15
2 assert(idade >= 18, "Você precisa ser +18 pra entrar!")
3 --> Vai dar erro e fechar o programa.
```

# error

**Não testa nada. Apenas encerra o programa com a mensagem de erro escolhida.**

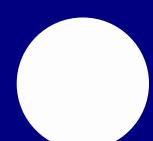
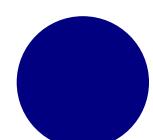
```
1 print('Como a vida é linda!!!')
2 error('ACABOU a gracinha desse programa')
3 print('Mas eu queria tanto executar :( )')
```

**não vai ser executado**





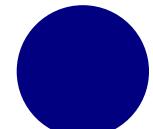
**Chegamos na  
parte avançada  
de Lua**



# **Metatabelas**

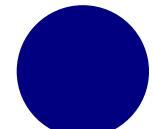
**Crie duas tabelas, da forma que quiser,  
com os elementos que quiser.**

**Tente somar elas e imprimi-las.**



# Metatabelas

Isso é um comportamento **indefinido!**  
Como não tem como saber **o que o usuário “quer dizer”** quando ele quer “somar” duas tabelas, o programa por padrão **dá erro.**



# Metatabelas

Metatabelas servem para **definir o comportamento** quanto às **operações** relacionadas à **tabelas** em Lua, principalmente quando essas operações são indefinidas por padrão, provendo extrema **customização** ao programador.

- “Similar” aos dunder methods do Python.

# Metatabelas e Metamétodos

**Metatabelas** são tabelas comuns que possuem metamétodos.

**Metamétodos** são formas de definir comportamentos quanto à operações envolvendo **tabelas**, principalmente através de **funções**.

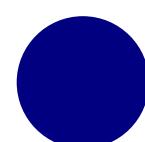


# Metatabelas e Metamétodos

**Metatabelas** são tabelas comuns que possuem **metamétodos**.

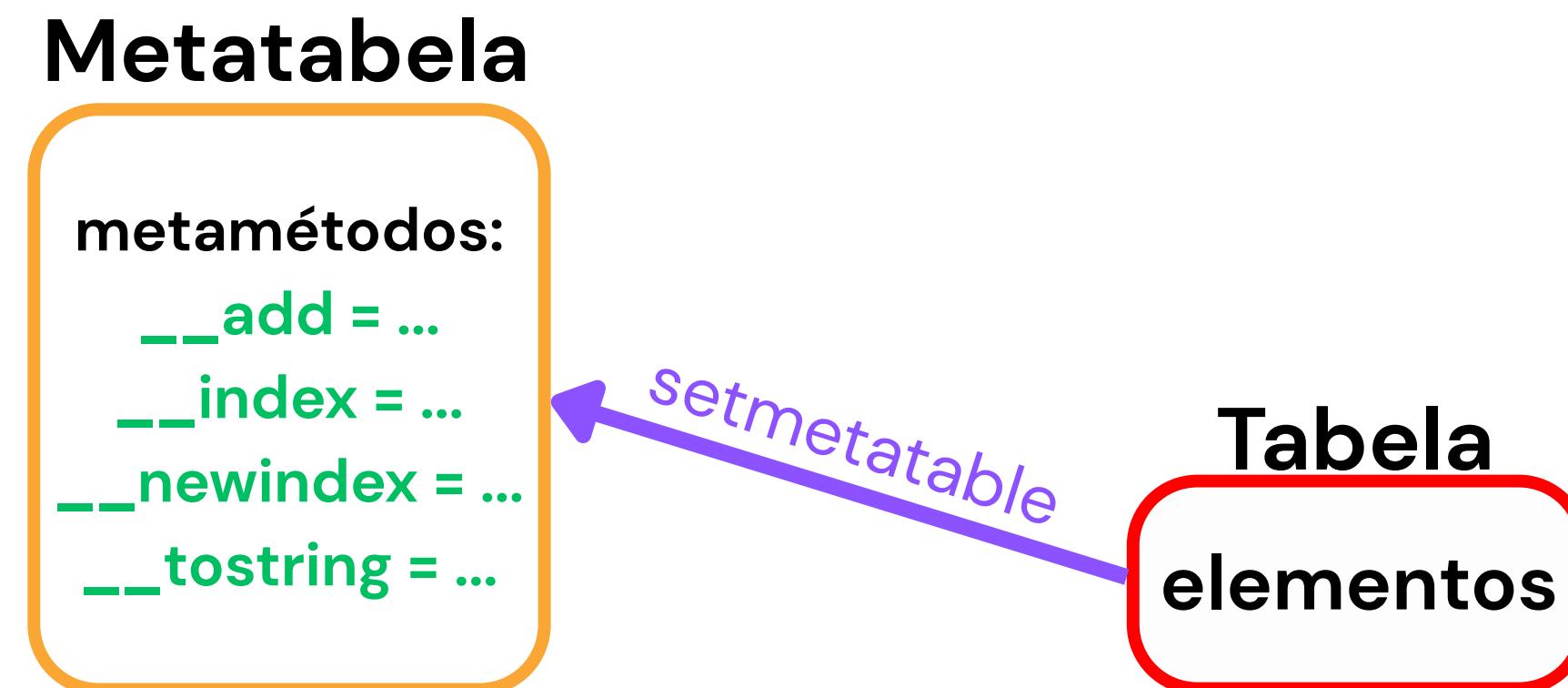
**Metamétodos** são formas de definir comportamentos quanto à operações envolvendo tabelas, principalmente através de **funções**.

Para definirmos sobre quais **tabelas** queremos que esses **metamétodos** atuem, atribuiremos uma **metatabela** (que contém **metamétodos**) sobre essas **tabelas**. Faremos isso com a função:  
**setmetatable(tabela, metatabela)**



# Metatabelas e Metamétodos

Diagrama:



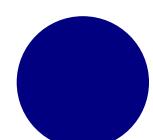
# Metatabelas e Metamétodos

Exemplo:

```
1 local numeros = {15, 30, 45, 60}
2 local outros_numeros = {200, 300, 400, 500}
3
4 local soma = numeros + outros_numeros
```

O que essa soma de tabelas era pra significar?

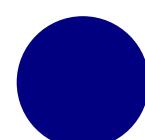
É pra somar todos os elementos da primeira  
tabela com todos os elementos da segunda? É pra  
multiplicar os elementos? É pra dar erro mesmo?



# Metatabelas e Metamétodos

Para **operações** entre **tabelas**, Lua irá procurar pra ver se **alguma** das **tabelas** tem algum **metamétodo** que define seu **comportamento** (pode ser só uma) .

Se existir, o comportamento da operação então será **este**.



# Metatabelas e Metamétodos

Exemplo:

```
4 local mt = {  
5     __add = function(tabela1, tabela2)  
6         local tab_result = {}  
7         for indice, valor in ipairs(tabela1) do  
8             tab_result[indice] = tabela1[indice] + tabela2[indice]  
9         end  
10        return tab_result  
11    end  
12}  
13 setmetatable(numeros, mt)
```

# Metatabelas e Metamétodos

Exemplo:

```
4 local mt = {  
5     __add = function(tabela1, tabela2)  
6         local tab_result = {}  
7         for indice, valor in ipairs(tabela1) do  
8             tab_result[indice] = tabela1[indice] + tabela2[indice]  
9         end  
10        return tab_result  
11    end  
12 }
```

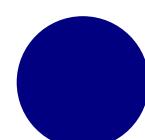
Criamos uma tabela 'mt' e nela adicionamos o metamétodo `__add`.

Todos os metamétodos são palavras reservadas sobre chaves de tabelas em Lua.

# Metatabelas e Metamétodos

**Exemplo:**

O que o **metamétodo `__add`** faz é: ele pode ser uma **função** que recebe até **2 parâmetros** (os dois operandos da adição), sendo o primeiro operando sempre a **tabela** que tem como **metatabela** a **tabela** que tem **`__add`** como **metamétodo**. Como uma adição entre dois elementos deve retornar algo, essa função então **retorna** o que é pra ser o **resultado** da adição.



# Metatabelas e Metamétodos

Exemplo:

```
1 local numeros = {15, 30, 45, 60}
```

```
13 setmetatable(numeros, mt)
```

Define 'mt' como  
a metatable de  
'numeros'



Mas pra esse **metamétodo** funcionar sobre a **tabela** desejada, devemos definir essa **tabela** como possuindo uma **metatable** com esse **metamétodo** **\_\_add!** (nesse caso)

# Metatabelas e Metamétodos

Agora a soma de  
**tabelas** que tem **mt**  
como metatable está  
definida pra operar na  
forma como foi  
designado em seu  
**metamétodo \_\_add!**

```
1 local numeros = {15, 30, 45, 60}
2 local outros_numeros = {200, 300, 400, 500}
3
4 local mt = {
5     __add = function(tabela1, tabela2)
6         local tab_result = {}
7         for indice, valor in ipairs(tabela1) do
8             tab_result[indice] = tabela1[indice] + tabela2[indice]
9         end
10        return tab_result
11    end
12 }
13 setmetatable(numeros, mt)
14
15 local soma = numeros + outros_numeros
16
17 for indice, valor in ipairs(soma) do
18     print(indice, valor)
19 end
```

# Metatabelas e Metamétodos

Podemos também definir o comportamento do metamétodo `__add` para dar **erro**.

```
1 local numeros = {15, 30, 45, 60}
2 local outros_numeros = {200, 300, 400, 500}
3
4 local mt = {
5     __add = error("Quero que nenhuma tabela seja somada não.")
6 }
7 setmetatable(numeros, mt)
8
9 local soma = numeros + outros_numeros...
10 --[[ 
11     como 'numeros' tem 'mt' como metatable
12     e essa metatable tem um metamétodo __add
13     definido, vai chamar ele pra definir o
14     comportamento dessa soma.
15 ]]
```

# Metatabelas e Metamétodos

Podemos também definir o comportamento para a soma de uma **tabela** e um **número**!

Um **if** com a função **type** é útil nesses casos específicos.

```
5 local mt = {  
6     __add = function(tabela1, segundo_operando)  
7         local tab_result = {}  
8  
9         if type(segundo_operando) == 'table' then  
10            for indice, valor in ipairs(tabela1) do  
11                tab_result[indice] = tabela1[indice] + segundo_operando[indice]  
12            end  
13  
14            return tab_result  
15        elseif type(segundo_operando) == 'number' then  
16            for indice, valor in ipairs(tabela1) do  
17                tab_result[indice] = tabela1[indice] + segundo_operando  
18            end  
19  
20            return tab_result  
21        end  
22    end  
23 }
```

# Metatabelas e Metamétodos

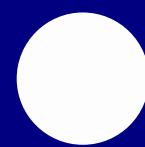
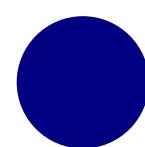
Em Lua, por padrão, ao criar tabelas elas não tem nenhuma metatable. Tabelas só terão um metatable se você mesmo fizer isso com a função: **setmetatable(tabela, metatable)**.

Podemos verificar qual a metatable de uma tabela com a função: **getmetatable(tabela)**.

```
local numeros = {15, 30, 45, 60}
local mt = {} -- metamétodos ficam aqui
setmetatable(numeros, mt)
print( getmetatable(numeros) ) -- retorna 'mt'
```

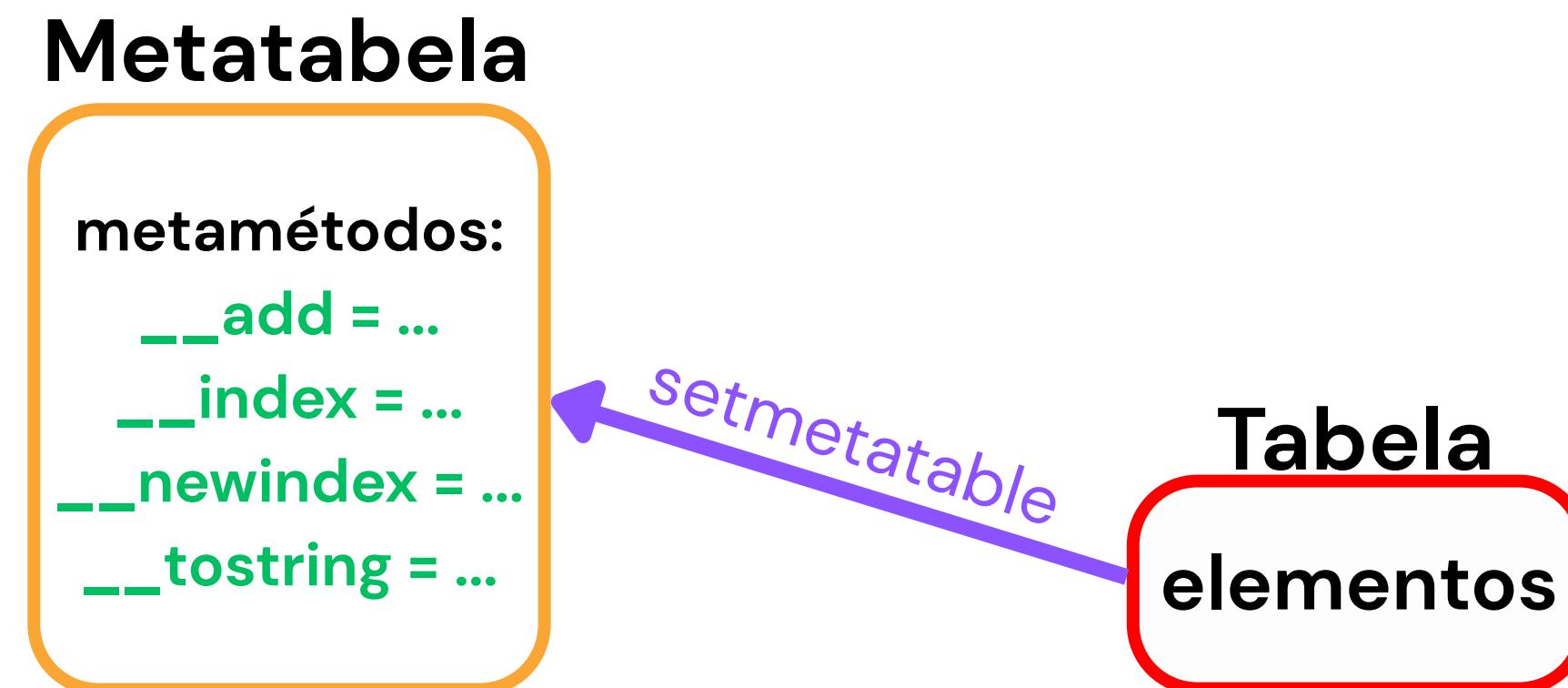
# Metatabelas e Metamétodos

Outros tipo de valores, como `number`, `string`, etc, também podem ter **metamétodos** aplicados sobre eles, mas isso só é possível com a **biblioteca debug** e é mais raro de encontrar essas implementações. Iremos aplicar **metatabelas e metamétodos** apenas sobre **tabelas** em virtude de ser bem mais comum.



# Metatabelas e Metamétodos

Diagrama:

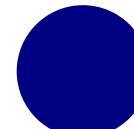


# Metatabelas e Metamétodos

**Lista de metamétodos que funcionam como `__add`:**

`__add`: operação de adição (+)  
`__sub`: operação de subtração (-)  
`__mul`: operação de multiplicação (\*)  
`__div`: operação de divisão (/)  
`__mod`: operação de módulo (%)  
`__pow`: operação de exponenciação (^)  
`__unm`: operação de negação unária (-)  
`__idiv`: operação de divisão de piso (//)  
`__lt`: operação de menor que (<)

`__band`: operação de bitwise AND (&)  
`__bor`: operação de bitwise OR (|)  
`__bxor`: operação de bitwise OR exclusivo (~)  
`__bnot`: operação de bitwise NOT unário (~)  
`__shl`: operação de bitwise left (««)  
`__shr`: operação de bitwise right (»»)  
`__concat`: operação de concatenação (..)  
`__eq`: operação de igual (==)  
`__le`: operação de menor ou igual que (<=)

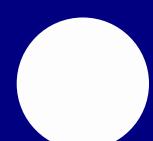
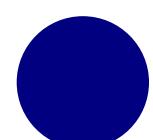


# Metatabelas e Metamétodos

Metamétodos diferentes:

**\_\_len:** define o comportamento pra quando chamamos o operador de tamanho (#) em uma tabela

**\_\_tostring:** define o comportamento pra quando a função `tostring` é chamada na tabela (a função `print` usa automaticamente a função `tostring` em tudo que ela imprime no terminal)



# Metatabelas e Metamétodos

```
1 local coisas = {"Sol", "Lua", "Marte", "Saturno"}  
2 local mt = {  
3     __len = function(tabela)  
4         local quantasLetras = 0  
5         for indice, valor in ipairs(tabela) do  
6             quantasLetras = quantasLetras + #valor  
7         end  
8         return quantasLetras  
9     end,  
10    __tostring = function(tabela)  
11        return "O total de letras de cada elemento da sua tabela somado é " .. #tabela  
12    end  
13}  
14 setmetatable(coisas, mt)  
15 print(coisas)
```



# Metatabelas e Metamétodos

Metamétodos diferentes:

**\_\_call:** permite chamar uma tabela como se fosse uma função

```
1  local numeros = {10, 20, 30}
2
3  local mt = {
4      __call = function(tabela, parametro)
5          for indice, valor in ipairs(tabela) do
6              tabela[indice] = tabela[indice] * parametro
7          end
8      end
9  }
10
11 setmetatable(numeros, mt)
12 numeros(5) --> multiplica todos os elementos por 5
```

# Metatabelas e Metamétodos

## METAMÉTODOS IMPORTANTES:

**`__index`: Quando procuramos um índice/chave em uma tabela e esse índice/chave não está presente na tabela, se essa tabela possui uma metatable com o metamétodo `__index`, `__index` irá definir onde que essa tabela deve procurar por um `__index`.**



Basicamente uma resposta pra pergunta: Qual vai ser o valor nesse índice/chave que não existe que você chamou? Eu devo tirar esse valor de outra tabela que tem esse índice que você chamou?

**será útil pra programação orientada a objetos!**

# Metatabelas e Metamétodos

`__index:`

```
1 local informacoes = {Altura = "1.61cm", Peso = "71kg", Idade = "25"}
2 local OUTRAS_infos = {Profissao = "Desenvolvedora", Cidade = "Bagé"}
3
4 local mt = {__index = OUTRAS_infos}
5
6 print(informacoes.Altura, informacoes.Peso, informacoes.Idade) --> funciona normal
7
8 print(informacoes.Profissao)      --> retorna nil, não tem como achar ainda
9
10 setmetatable(informacoes, mt)
11
12 print(informacoes.Profissao)      --> agora funciona pois usa __index pra achar
```

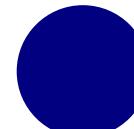
# Metatabelas e Metamétodos

## METAMÉTODOS IMPORTANTES:

**\_\_newindex**: Define o que vai acontecer quando tentamos atribuir algum valor para um índice/valor da tabela cujo valor era nil:

`tabela[índice_que_não_existia] = alguma_coisa`

**será útil pra programação orientada a objetos!**



# Metatabelas e Metamétodos

**\_\_newindex:**

```
1 local prototipoPessoa = {Nome = "", Idade = 0, Profissao = "", NasceuEm = ""}
2 local gabriel = {}
3 local mt = {
4     __newindex = function(tabela, indice, valor)
5         if prototipoPessoa[indice] == nil then
6             error("Isso não existe no protótipo de uma pessoa!")
7         else
8             rawset(tabela, indice, valor)
9         end
10    end
11 }
12 setmetatable(gabriel, mt)
13
14 gabriel.Nome = "Gabriel" --> Funciona!
15 gabriel.PodeSairVoandoDoNada = true --> NÃO FUNCIONA!
```

# Metatabelas e Metamétodos

`__newindex`: Usaremos **rawset** para não causar recursão infinita (o que ocorreria se tentássemos atribuir um valor a um novo index novamente).

`rawset(tabela, indice, valor)`



mesma coisa que:  
`tabela[index] = value`

É apenas  
um escape!

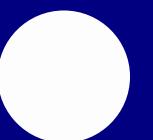
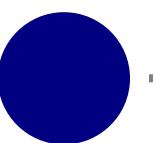


# **Metatabelas e Metamétodos**

## **Exercício Prático nº 5**

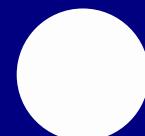
**Esse é livre! Escolha um ou mais *metamétodos* e use eles para testar seu entendimento dos seus conceitos.**

**Busque trabalhar entre ou sobre tabelas e definindo comportamentos.**



# Programação Orientada a Objetos

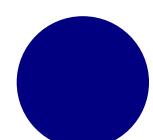
Imagine que você está fazendo um jogo RPG e você quer que seja possível existirem **vários personagens** que sejam **magos** no jogo. Esses **magos** podem usar **feitiços**, usar **poderes especiais** e tudo mais. Como você implementaria isso? Você iria definir várias vezes, por exemplo, uma mesma função **usarPoder()**, uma pra cada **mago**? Mesmo que essa função fizesse sempre a mesma coisa?



# Programação Orientada a Objetos

Você até poderia fazer isso, mas seria muito entediente e tomaria muito do seu tempo! O ideal seria que você só tivesse que **escrever esse código uma vez** e utilizar esse **mesmo código** para **todos os magos**, certo?

Isso é exatamente sobre o que **Programação Orientada a Objetos** se trata!



# Programação Orientada a Objetos

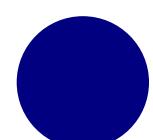
## Classes e Objetos:

**Classe:** É o protótipo de um **objeto**. Possui todos os **valores e funções (métodos)** que o nosso objeto irá utilizar.

**Objeto:** É uma instância de uma **classe**. Vai possuir atributos de acordo com como a classe definir ser possível e pode utilizar dos métodos de sua classe.

Se a classe é o plano de uma casa, a instância é uma casa construída com base nesse plano.

Perceba que é completamente possível várias casas seguirem o mesmo plano.



# Programação Orientada a Objetos

Pseudo-código de uma classe:

```
classe X {  
    Nome = " ", Idade = 0, Dinheiro = 0,  
    dizOi = print("oiii"),  
    construtor(Nome, Idade, Dinheiro)  
}
```

Atributos →

Nome = " ", Idade = 0, Dinheiro = 0,

Métodos →

dizOi = print("oiii"),

Construtor →

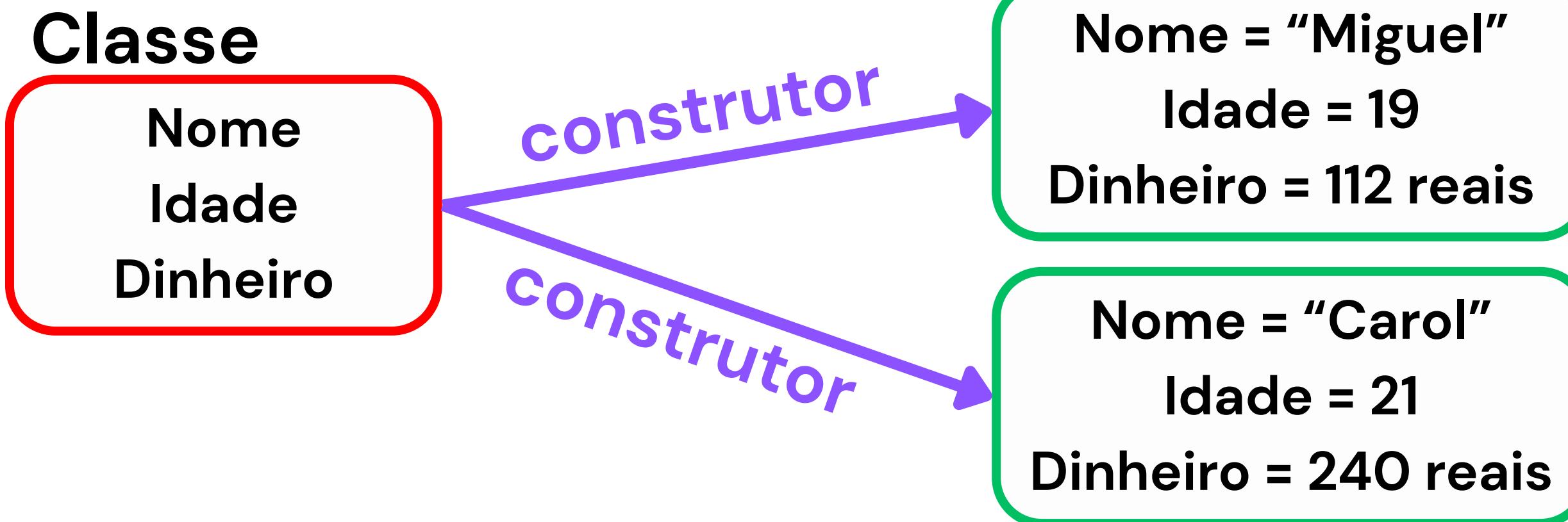
construtor(Nome, Idade, Dinheiro)

Atributos não possuem valores na classe pois eles só vão ter valores nos objetos dessa classe!



# Programação Orientada a Objetos

A função construtor é para criar objetos de uma classe:



## Objetos

O **construtor** permite os **objetos** terem seus **valores**, dados os atributos possíveis da **classe**.

# Programação Orientada a Objetos

Existem várias formas de implementar **orientação a objetos** em **Lua**, vamos ver uma das interpretações (**talvez a mais famosa**).



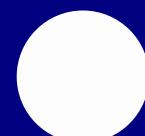
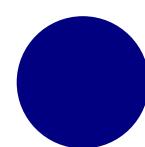
# Programação Orientada a Objetos

Como fazer em Lua: (Exemplo Conta Bancária)

```
4 Conta_Bancaria = {  
5     dono = "",  
6     idade = 0,  
7     moradia = "",  
8     saldo = 0,  
9 }
```

Definindo os **atributos** da  
**classe Conta\_Bancaria**

**Classes** em Lua serão  
representadas por **tabelas**.

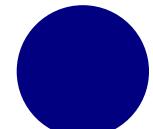


# Programação Orientada a Objetos

Essa parte é meio confusa, escute com atenção:

Como **classes** são uma **tabela** em Lua, os **objetos** também serão. Os **objetos** deverão ter como **atributos** possíveis apenas aqueles da **classe** que eles são.

Essencialmente, queremos copiar as **chaves (atributos)** da tabela **classe** para o **objeto**, atribuindo, no **objeto**, valores para esses **atributos**. Também queremos definir o comportamento das **chaves** do **objeto** para que só seja possível existirem **chaves (atributos)** iguais os da **classe**!



# Programação Orientada a Objetos

Definir comportamento? → Metatabelas!

`__newindex`: Define o que vai acontecer quando tentamos atribuir algum valor para um índice/valor da tabela cujo valor era nil:

```
tabela[índice_que_não_existia] = alguma_coisa
```

`__index`: Quando procuramos um índice/chave em uma tabela e esse índice/chave não está presente na tabela, se essa tabela possui uma metatable com o metamétodo `__index`, `__index` irá definir onde que essa tabela deve procurar por um `__index`.

Queremos que índices/chaves/atributos novos sejam proibidos!

Queremos que os objetos procurem índices/chaves/atributos na tabela classe!

# Programação Orientada a Objetos

Além dos **atributos**, a tabela possuirá **metamétodos**, no intuito de virar a **metatabela** do **objeto**. Isso é para que o **objeto**, por causa do **\_\_index = Conta\_Bancaria**, procure **atributos** na **classe** e, por causa do **\_\_newindex** estar definido dessa forma, **objetos** não poderão ter outros **atributos**!

```
11 Conta_Bancaria.__index = Conta_Bancaria
12
13 Conta_Bancaria.__newindex = function(objeto, atributo, valor)
14     if Conta_Bancaria[atributo] == nil then --> esse atributo existe na tabela classe? não existe=erro, existe=atribui normalmente
15         error("Não pode adicionar mais atributos em um objeto da classe Conta_Bancaria!", 2)
16     else
17         rawset(objeto, atributo, valor)
18         --> deve-se usar rawset pois atribuição comum (objeto[atributo] = valor) causa recursão infinita! (fica sempre chamando o __newindex)
19         --> rawset faz a mesma coisa que a atribuição comum, ele serve apenas nessas situações como um "escape" (ele não usa o __newindex, faz direto)
20     end
21 end
```

# Programação Orientada a Objetos

Essa função da **tabela-classe**

**Conta\_Bancaria** é seu **construtor!**

Normalmente chamamos a função

**construtor de new.**

Essa função recebe os **atributos** que um  
objeto dessa **classe** terá pelos  
**parâmetros**.

É criada uma nova tabela objeto, que  
terá a sua **classe** (com todos aqueles  
metamétodos) como **metatabela**. Ao  
final, o **objeto** é retornado para ser  
utilizado normalmente no código.

```
32 Conta_Bancaria.new = function(dono, idade, moradia)
33   local objeto = setmetatable({}, Conta_Bancaria)
34
35   objeto.dono      = dono
36   objeto.idade    = idade
37   objeto.moradia  = moradia
38
39   return objeto
40 end
```

# Programação Orientada a Objetos

Agora com o **construtor** podemos criar vários **objetos** da **classe Conta\_Bancaria!**

O **construtor .new** é chamado, na **função** criam uma **tabela objeto**, passam a **tabela-classe Conta\_Bancaria** como **metatabela do objeto**, e essa faz com que os atributos sejam os mesmos da **classe**. Os **valores** dos atributos do **objeto** (agora iguais aos da **classe**) serão os que foram passados pro **construtor**. O **objeto** é retornado. Agora a **instância do objeto** é uma tabela com os atributos da **classe** com valores.

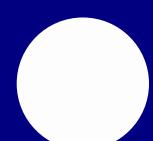
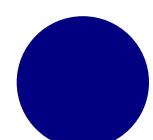
```
56      -----> main <-----  
57  
58  local conta_do_jairo = Conta_Bancaria.new("Jairo", 49, "Rua Gomes Carneiro, 1")  
59  local conta_da_bruna = Conta_Bancaria.new("Bruna", 22, "Rua XV de Novembro, 598")  
60  local conta_do_naldo = Conta_Bancaria.new("Naldo", 36, "Rua Cel. Alberto Rosa, 154")  
61
```

# Programação Orientada a Objetos

Agora só faltam os **métodos** (não confundir com metamétodos), as **funções** da **classe**, que serão utilizadas pelos **objetos**. Bom, como essas **funções/métodos** serão utilizadas por cada objeto, já que, nesse nosso exemplo da Conta Bancária, precisamos especificar qual **objeto** está fazendo um **depósito** em sua conta, já que nesse caso o **objeto** representaria uma pessoa.

```
56     -----> main <-----  
57  
58     local conta_do_jairo = Conta_Bancaria.new("Jairo", 49, "Rua Gomes Carneiro, 1")  
59     local conta_da_bruna = Conta_Bancaria.new("Bruna", 22, "Rua XV de Novembro, 598")  
60     local conta_do_naldo = Conta_Bancaria.new("Naldo", 36, "Rua Cel. Alberto Rosa, 154")  
61
```

O depósito vai ser pro Jairo, Bruna ou Naldo?



# Programação Orientada a Objetos

Aqui está um jeito de fazer um **método/função** da **classe Conta\_Bancaria**. Esse **método** deverá operar sobre o **objeto** que chamar esse **método**, já que queremos que ele afete apenas a conta bancária desse objeto (nesse caso uma pessoa) e não de todos os **objetos/pessoas**.

Então passamos como **parâmetro** 'esse\_objeto' que será a **tabela do objeto**, e pra isso funcionar teremos que passar a **própria tabela de um objeto** como argumento pra esse **método** no código principal.

```
42 function Conta_Bancaria.depositar(esse_objeto, quantia)          (Na definição do método na classe)
43     esse_objeto.saldo = esse_objeto.saldo + quantia
44     print("Depósito de " .. quantia .. " realizado, tens agora " .. esse_objeto.saldo .. " reais!")
45 end
```

No código principal:

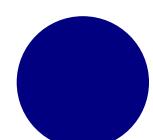
```
61     conta_do_jairo.depositar(conta_do_jairo, 567)
```

# Programação Orientada a Objetos

```
61     conta_do_jairo.depositar(conta_do_jairo, 567)
```

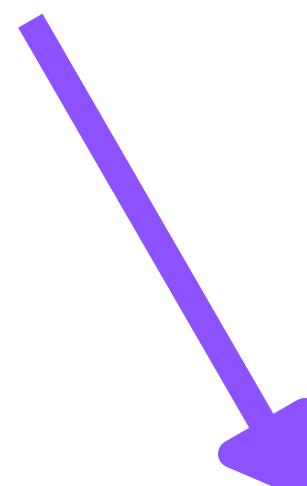
Percebe como estamos tendo que escrever o nome do **objeto duas vezes**? Uma vez pra definir **qual objeto** queremos usar o **método** e outra pra definir **qual tabela** será passada para esse método? Não deveria ser óbvio que queremos passar o **objeto** que estamos trabalhando como parâmetro pra esse **método**?

É aí que o **syntax sugar** (açúcar sintático) do **'.'** e do **self** entram em cena!



# Programação Orientada a Objetos

```
61 conta_do_jairo.depositar(conta_do_jairo, 567)
```



Ao utilizar ‘:’ na **chamada do método** entre o nome do **objeto** e o nome do **método**, **automaticamente** passamos a **tabela do objeto** que estamos trabalhando como **primeiro argumento->parâmetro** do **método!**

```
61 conta_do_jairo:depositar(567)
```



# Programação Orientada a Objetos

```
42 function Conta_Bancaria.depositar(esse_objeto, quantia)
43     esse_objeto.saldo = esse_objeto.saldo + quantia
44     print("Depósito de " .. quantia .. " realizado, tens agora " .. esse_objeto.saldo .. " reais!")
45 end
```

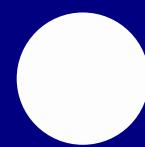
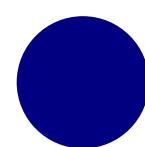


Podemos fazer o mesmo com a definição do método! Ao colocar ':' entre o nome da **classe** e o nome do **método**, automaticamente estamos passando uma **instância de objeto** chamada "**self**" em Lua como **primeiro parâmetro da função**, sem termos que adicionar!

```
42 function Conta_Bancaria:depositar(quantia)
43     self.saldo = self.saldo + quantia
44     print("Depósito de " .. quantia .. " realizado, tens agora " .. self.saldo .. " reais!")
45 end
```

# **Programação Orientada a Objetos**

E isso é tudo de **programação  
orientada a objetos em Lua!**



# Programação Orientada a Objetos

Outro método de exemplo:

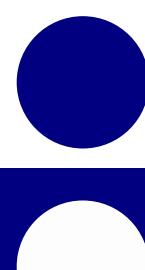
```
47 function Conta_Bancaria:retirar(quantia)
48     if self.saldo < quantia then
49         print("Você só tem " .. self.saldo .. " de saldo! Não dá pra retirar " .. quantia .. " reais.")
50     else
51         self.saldo = self.saldo - quantia
52         print("Você retirou " .. quantia .. " reais, tens agora " .. self.saldo .. " reais!")
53     end
54 end
```

# Programação Orientada a Objetos

Adicionando o **metamétodo `__tostring`**

podemos mudar o `print` de um objeto

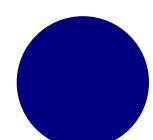
```
23 Conta_Bancaria.__tostring = function(obj)
24     return "-----BANCO-----\n ...
25     "Dono\t= " ... obj.dono ...
26     "\nIdade\t= " ... obj.idade ...
27     "\nMoradia\t= " ... obj.moradia ...
28     "\nSaldo\t= " ... obj.saldo ...
29     "\n-----"
30 end
```



# Programação Orientada a Objetos

## Exercício Prático nº 6

Escolha alguma situação qualquer para criar utilizando **programação orientada a objetos** em **Lua** seguindo o exemplo da Conta Bancária. Ideias: **Personagens de jogos e seus poderes**, **Carros e Movimento**, **Animais e suas Ações**, **Posições, Vetores e Álgebra Linear**, etc.



# Programação Orientada a Objetos

Podemos também implementar Herança ao fazer com que nossas metatabelas (com metamétodo `__index`) também tenha metatabelas com metamétodos `__index`)



# Corrotinas

**Corrotinas** são simplesmente funções em Lua que podem ter sua execução **parada** e **resumida** a qualquer momento na execução do programa.



# Corrotinas

Para utilizar corrotinas precisamos da tabela coroutine

**coroutine.create** → cria uma corrotina

**coroutine.resume** → resume (ou inicia) a execução de uma corrotina

**coroutine.yield** → pausa a execução de uma corrotina

**coroutine.status** → retorna a situação da corrotina



# Corrotinas

```
1 local co = coroutine.create(  
2     function()  
3         for i=1, 5 do  
4             print(i)  
5             coroutine.yield()  
6         end  
7     end  
8 )  
9  
10 print(coroutine.status)  
11 coroutine.resume(co)      --> inicia pela 1º vez  
12 coroutine.resume(co)  
13 coroutine.resume(co)  
14 coroutine.resume(co)  
15 coroutine.resume(co)      --> última execução, função acabou  
16 print(coroutine.status)
```

# Corrotinas

**yield pode  
retornar  
valores e  
resume  
pode enviar  
valores**

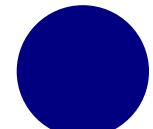
# Corrotinas

**Corrotinas** em Lua não são paralelas. Na **prática** elas servem apenas para poder pausar o funcionamento de uma **função** para utilizar **outra**.



# API Lua-C

Para a API funcionar, Lua e C  
devem ser da mesma  
**arquitetura** em nosso  
computador, por exemplo,  
ambas serem **64 bits**.



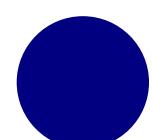
# API Lua-C

Para a API funcionar,  
**Lua** e **C** devem ser da  
mesma **arquitetura** em  
nosso computador,  
por exemplo, ambas  
serem **64 bits**.

É um problema comum, ao usar  
**MinGW**, que sua **arquitetura** do **C** seja  
**32 bits**. As versões mais modernas de  
**Lua** são **64 bits**. Isso gera um **conflito**.  
Execute no terminal para verificar sua  
arquitetura:  
**C**: `gcc -dumpmachine`  
**Lua** (no REPL): `print(8 * string.packsize('j'))`

# API Lua-C

É bom adicionar o path para o IntelliSense encontrar os header files da biblioteca Lua. Pra isso podemos, no VS Code, clicar Ctrl+Shift+P e pesquisar “C/C++ UI” e editar. Vá em Incluir Caminho e coloque numa nova linha o caminho para a pasta ‘include’ da sua pasta Lua (a compilada)



# API Lua-C

Quando usamos a API Lua-C em C, precisamos incluir os header files no topo do nosso programa:

**#include <lua.h>** → inclui funções para criar ambiente Lua, chamar funções Lua, ler e escrever variáveis globais nesse ambiente, registrar novas funções para serem chamadas, etc

**#include <lauxlib.h>** → inclui funções providas pela biblioteca auxiliar de lua.

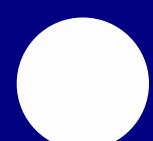
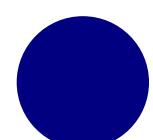
**#include <lualib.h>** → declara funções para abrir as bibliotecas.

**“luaL\_openlibs” abre todas as bibliotecas padrão**

# API Lua-C

Todas as **funções** do **header file** **lua.h** **começam**  
**com** (tem prefixo) **'lua\_'**

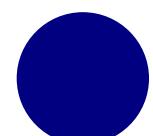
Todas as **funções** dos **header files** **lauxlib.h** e  
**lualib.h** **começam com** (tem prefixo) **'luaL\_'**



# API Lua-C

header files (.h):

- São arquivos com **declarações** de funções, macros, tipos, structs etc.
- Eles **não implementam** (executam) as funções, só **descrevem** como elas são.
- Eles ajudam o compilador a saber como gerar o código.



# API Lua-C

A implementação real fica com as bibliotecas:

- Os **header files** só declaram as funções, o **código real** (a **implementação das funções**) está na **biblioteca compilada**:
  - um **.dll** (no Windows),
  - um **.so** (no Linux)

# API Lua-C

Para além de incluir os **header files** no **código**, precisamos especificar na chamada do **compilador gcc** em **qual pasta** podemos encontrar esses **header files**, onde encontrar **outra pasta** que contém a **implementação (dll, so)** dessas **funções** declaradas por esses **header files**, e qual **biblioteca (dll, so)** queremos utilizar.



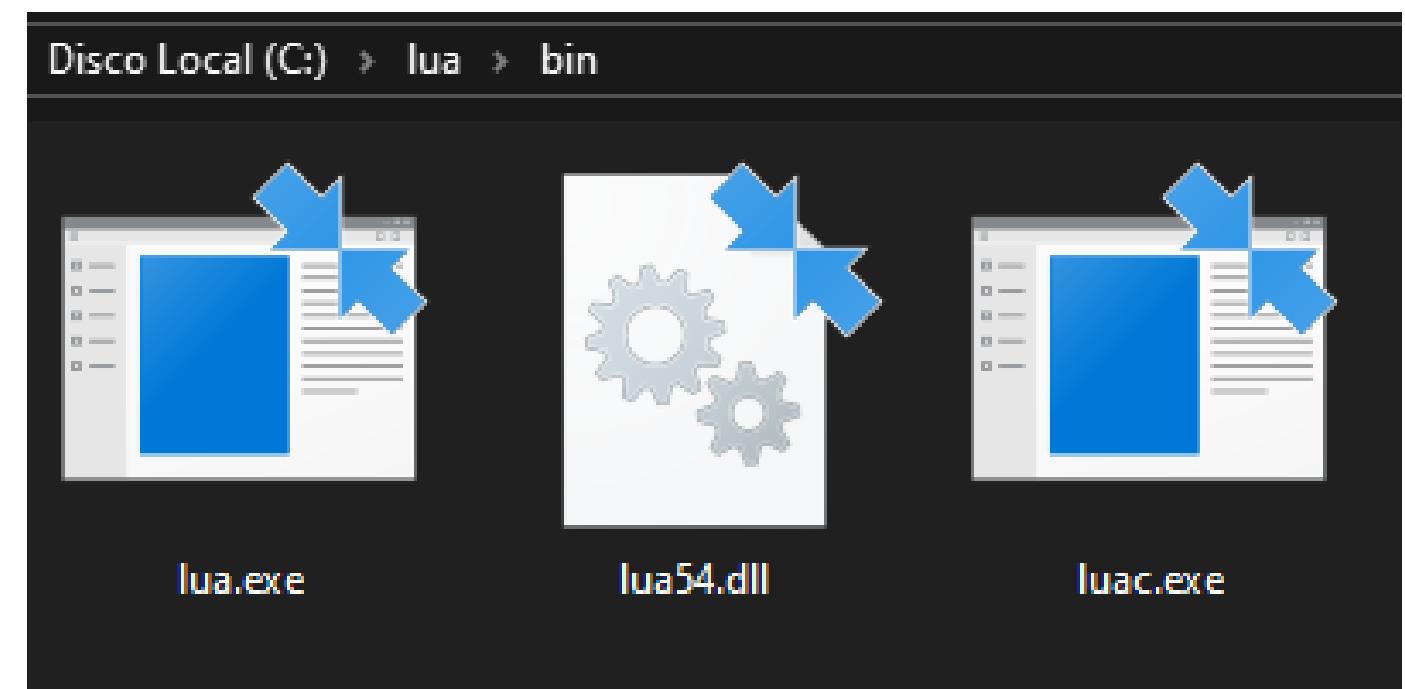
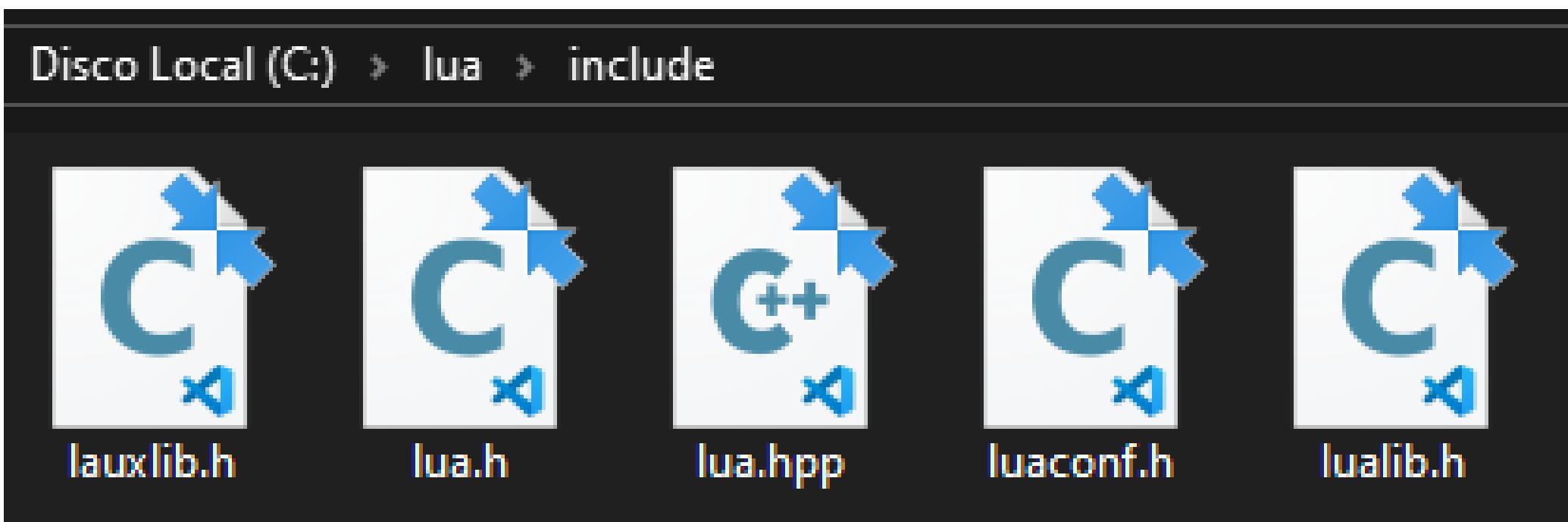
# API Lua-C

**gcc app.c -o app.exe -IC:/lua/include -LC:/lua/bin -llu54**

onde encontrar  
os headers files

onde encontrar  
as bibliotecas

qual biblioteca  
queremos usar



I é de link library

# API Lua-C - Lua Stack

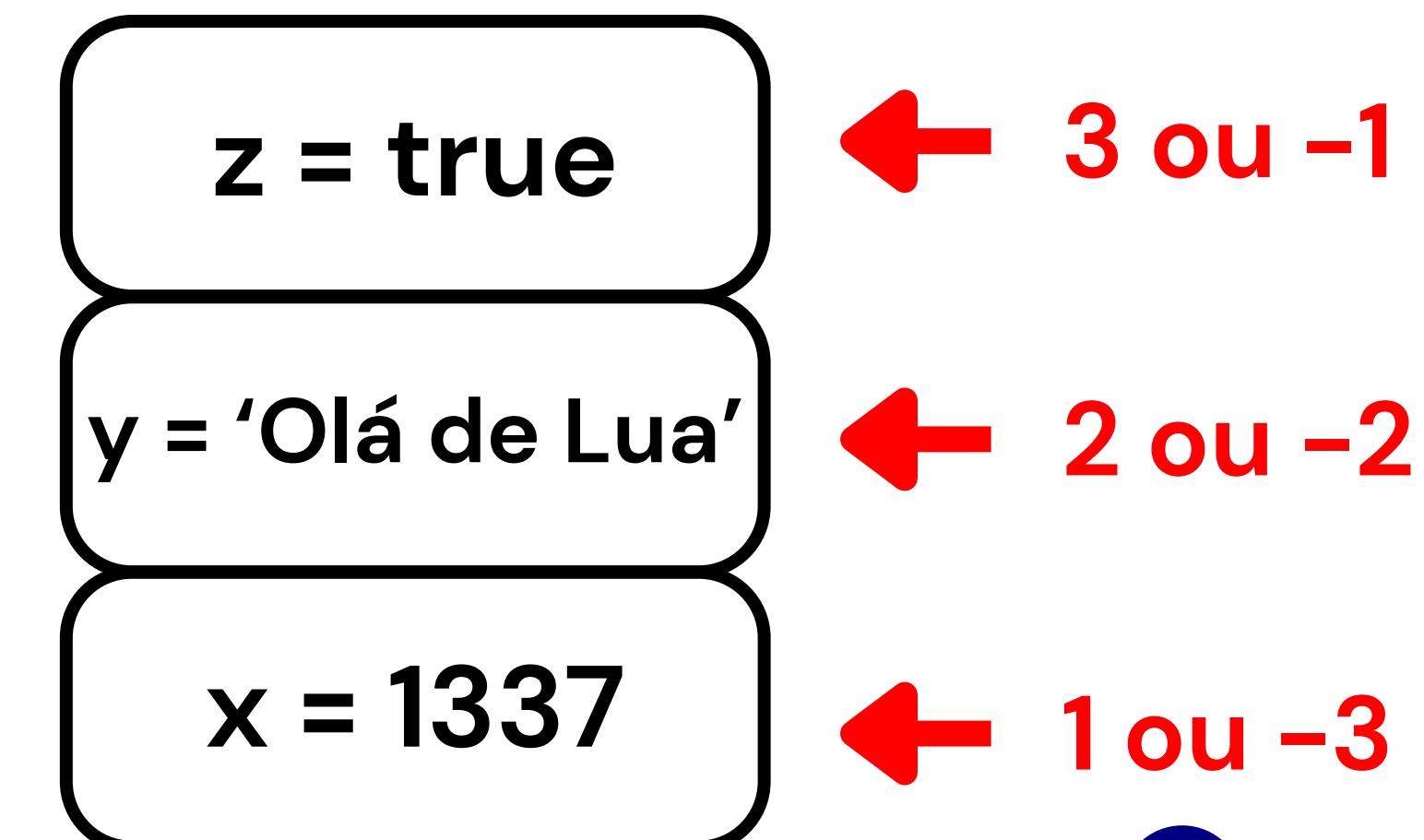
Todas as **variáveis globais** que definirmos em **Lua** serão adicionadas numa **stack** (**pilha**) de **valores** em ordem de criação no programa. Essa é a **Lua Stack**. A **Lua Stack** depois será passada para a **stack Lua-C** no nosso programa **C**.



# API Lua-C - Lua Stack

Digamos que nós temos esse arquivo lua:

```
lua: bomdia.lua > ...
1   x = 1337
2   y = "Olá de Lua"
3   z = true
```



# API Lua-C

Agora, em C, criamos um ponteiro \*L com `luaL_newstate()` para criar uma instância de Lua, abrimos suas bibliotecas com `luaL_openlibs(L)`, rodamos o nosso programa lua com `luaL_dofile(L, "nomeDoPrograma")`, pegamos as variáveis da Lua Stack com `lua_getglobal`, assim passando elas pra Stack Lua-C (stack do C), convertemos os valores dessa stack de valores Lua pra C com `lua_to"tipo"` e fechamos essa instância de Lua com `lua_close(L)`.

```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4 #include <lua.h>
5 #include <lualib.h>
6 #include <lauxlib.h>
7
8 int main(){
9
10    lua_State *L = luaL_newstate(); //inicializa um state de lua, uma stack Lua-C
11    luaL_openlibs(L);           // inicializa as bibliotecas padrão de lua
12
13    luaL_dofile(L, "bomdia.lua"); // abre o arquivo
14
15    lua_getglobal(L, "x");
16    lua_getglobal(L, "y"); // Adiciona as variáveis globais declaradas em Lua na stack Lua-C.
17    lua_getglobal(L, "z"); // Elas vão da (Stack Lua) -> (Stack Lua-C)
18
19    int x = lua_tonumber(L, 1);
20    const char* y = lua_tostring(L, 2); // atribui os valores da stack Lua-C para variáveis em C
21    bool z = lua_toboolean(L, 3);
22
23    printf("x = %d\ny = %s\nz = %d", x, y, z);
24
25    lua_close(L); // finaliza e libera esse state de Lua e sua stack
26
27    return 0;
28 }
```

# API Lua-C

```
int x = lua_tonumber(L, 1);
```

- Pega o primeiro valor da **stack** **Lua-C** (deve ser um número nesse caso)
- 1 significa que estamos pegando o **primeiro item** que foi **adicionado** na **Stack**

1 → primeiro item que foi adicionado  
-1 → último item que foi adicionado

```
int x = lua_tonumber(L, -3);
```

- Podemos também **utilizar valores negativos** para acessar a **Stack do topo** (nesse exemplo, sabemos que a stack tem 3 itens, como queremos o **primeiro item**, vamos usar **-3**)

# API Lua-C

`luaL_dofile(L, "nomeDoPrograma")`, além de tornar possível (não fazer, apenas tornar possível) conectar a **Stack Lua** com a **Stack Lua-C**, `luaL_dofile` realmente executa o programa, então se, por exemplo, no nosso programa “**bomdia.lua**”, além de **declarar as variáveis x, y e z**, chamarmos essa **função**:

```
6 print("Lua mandou oi!")
```

No nosso programa de exemplo anterior **app.c** onde fizemos `luaL_dofile(L, "bomdia.lua")`, se o **compilarmos e executarmos**, mesmo que `print` seja uma **função** de **Lua** e estamos em um **programa C**, ele será **executado pelo dofile!**



```
Lua mandou oi!  
x = 1337  
y = Olá de Lua  
z = 1
```

# API Lua-C

Chamando **funções** de **Lua** em **C**:

Digamos que nós temos essa **função** em **Lua**:

```
lua printXvezes.lua > ...
1   printMsg = function(mensagem, x)
2       for i=1, x do
3           print(mensagem)
4       end
5   end
```

A referência da **função** deve estar armazenada em uma **variável global**

# API Lua-C

Chamando **funções de Lua em C**:

Como passamos **2 argumentos** para a chamada de **função (lua\_call)**, ele irá na posição **-2** da **stack** (onde começam os parâmetros que adicionamos), então se pegará o valor **anterior** a esse na **stack** (no nosso caso **-3**) e esse valor será a função a ser chamada com os **parâmetros** na sequência da **stack**.

Os **parâmetros** são **removidos** da **stack** após o fim da execução da **função**, não dá pra usar de novo.

Caso exista valores de **retorno**, esses irão para a **stack**.

Outra alternativa a **lua\_call** é usar **lua\_pcall**, que é feito para trabalhar quando existem erros:

**lua\_pcall(L, 2, 0, 0) != 0** **deu erro!**

```
1 #include <lua.h>
2 #include <lauxlib.h>
3 #include <lualib.h>
4
5 int main(){
6
7     lua_State *L = luaL_newstate();
8     luaL_openlibs(L);
9
10    luaL_dofile(L, "printXvezes.lua");
11
12    lua_getglobal(L, "printMsg"); // adiciona a função na stack
13
14    // adiciona a string e o número na stack Lua-C \
15    lua_pushstring(L, "Vou colocar essa string na stack Lua-C!");
16    lua_pushnumber(L, 5);
17
18    lua_call(L, 2, 0);
19    // 2 significa que a função tem 2 argumentos e 0
20    // significa que a função tem 0 valores a serem retornados
21
22    luaL_close(L);
23    return 0;
24 }
```

# API Lua-C

Criando **tabelas** de Lua em **C**:

Imaginemos um exemplo similar ao anterior:

```
2 printTabela = function(tabela)
3     for i = 1, #tabela do
4         print(tabela[i])
5     end
6 end
```

A referência da  
**função** deve estar  
armazenada em  
uma **variável global**

# API Lua-C

Criando **tabelas** de Lua em **C**:

No caso de **tabelas** como **arrays**,  
simplesmente criamos um **array** em **C**,  
utilizamos a função **lua\_newtable(L)**  
para criar uma **tabela** na **Lua-C Stack** e  
usamos um **for-loop** para colocar o  
**índice** e o **valor** na **Lua-C Stack**, e a  
função **lua\_settable** pegará esses  
elementos e os adicionará na **tabela**,  
também removendo esses elementos  
da **stack**.

```
5  int main(){
6
7      lua_State *L = luaL_newstate();
8      luaL_openlibs(L);
9
10     luaL_dofile(L, "imprimeTabela.lua");
11
12     lua_getglobal(L, "printTabela"); // adiciona a função na stack
13
14     int numeros[5] = {10, 20, 30, 40, 50}; // serão os valores da tabela
15     lua_newtable(L); // cria nova tabela (novo elemento na stack)
16
17     for (int i = 0; i < 5; i++){
18         //adiciona a chave depois o valor na stack
19         lua_pushnumber(L, i+1); // i começa em 0 e stack começa em 1
20         lua_pushnumber(L, numeros[i]);
21         lua_settable(L, -3); // adiciona os dois últimos elementos na tabela
22         // depois de adicionados, esses elementos são removidos da stack
23     }
24
25     lua_call(L, 1, 0);
26     lua_close(L);
27     return 0;
28 }
```

# API Lua-C

Como criar **bibliotecas** em **C**  
para usar em **Lua** (**dll's, so's**)

```
1 #include <stdio.h>
2 #include <lua.h>
3 #include <lauxl.h>
4 #include <lualib.h>
5
6 int olaMundo(lua_State *L){
7     lua_pushstring(L, "Olá, mundo!");
8     return 1; // estamos retornando a string
9 }
10
11 int printOi(lua_State *L){
12     printf("Oi.");
13     return 0; // não estamos retornando nada
14 }
15
16 // isso vai virar uma tabela em Lua com as funções
17 // mapeia chaves (nome da função em Lua) -> valores (funções de C)
18 static const struct luaL_Reg funcoes[] = {
19     {"olaMundo", olaMundo},
20     {"printOi", printOi},
21     {NULL, NULL}    // é o "nil", a tabela acaba aqui -> necessário
22 };
23
24 // luaopen_Nome deve ter o nome MESMO da biblioteca
25 // cria uma tabela em lua com a "tabela" (array de structs) funcoes, que tem as funções
26 int luaopen_Minicurso(lua_State *L){
27     luaL_newlib(L, funcoes);
28     return 1;
29 }
```

# API Lua-C

Para compilar:

```
gcc 'nomeArquivo'.c -shared -fpic -o 'NomeLib'.dll -I"caminhoInclude" -L"CaminhoLib" -l"Lib"
```

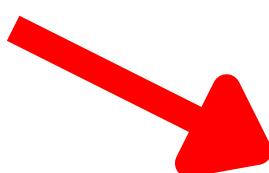
No meu caso:

```
gcc biblioteca.c -shared -fpic -o Minicurso.dll -IC:/lua/include -LC:/lua/bin -llu54
```

# API Lua-C

Agora em **Lua** (com a biblioteca na mesma pasta):

```
1 local bibliotecaC = require("Minicurso")
2
3 print(bibliotecaC.olaMundo())
4 bibliotecaC.printOi()
```



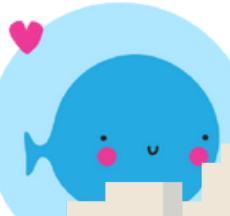
```
Olá, mundo!
Oi.
```



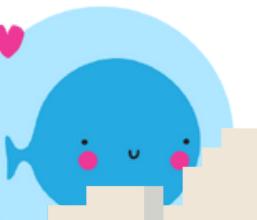
**E isso é tudo sobre Lua  
que iremos cobrir  
nesse Minicurso!**



**Bônus:**  
**Löve2D**



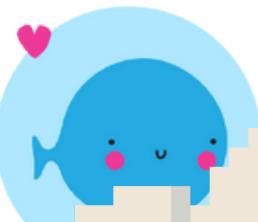
**Löve2D** é um framework de criação  
de jogos escrito principalmente em  
**C++** feito pra você programar seu  
jogo só usando **Lua!**



# Framework vs. Engine

**Um framework é um apenas um conjunto de ferramentas e bibliotecas pra auxiliar na criação de algo. (Conseguimos fazer jogos completos com Löve2D)**

**Uma engine é uma plataforma completa, geralmente com editor visual, gerenciamento de cenas, física, UI, ferramentas de debug, e muito mais.**



# Características

**Löve2D é royalty free**

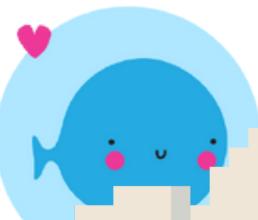
**É feito para criação de jogos 2D**

**Utiliza LuaJIT (ainda está na versão 5.1 de Lua)**

**É possível rodar jogos feitos nele em diversas plataformas,  
como Windows, Linux, Android, Nintendo Switch, etc...**

**Utiliza OpenGL**

**Possui tudo necessário para fazer um jogo comercial**

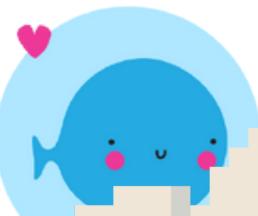
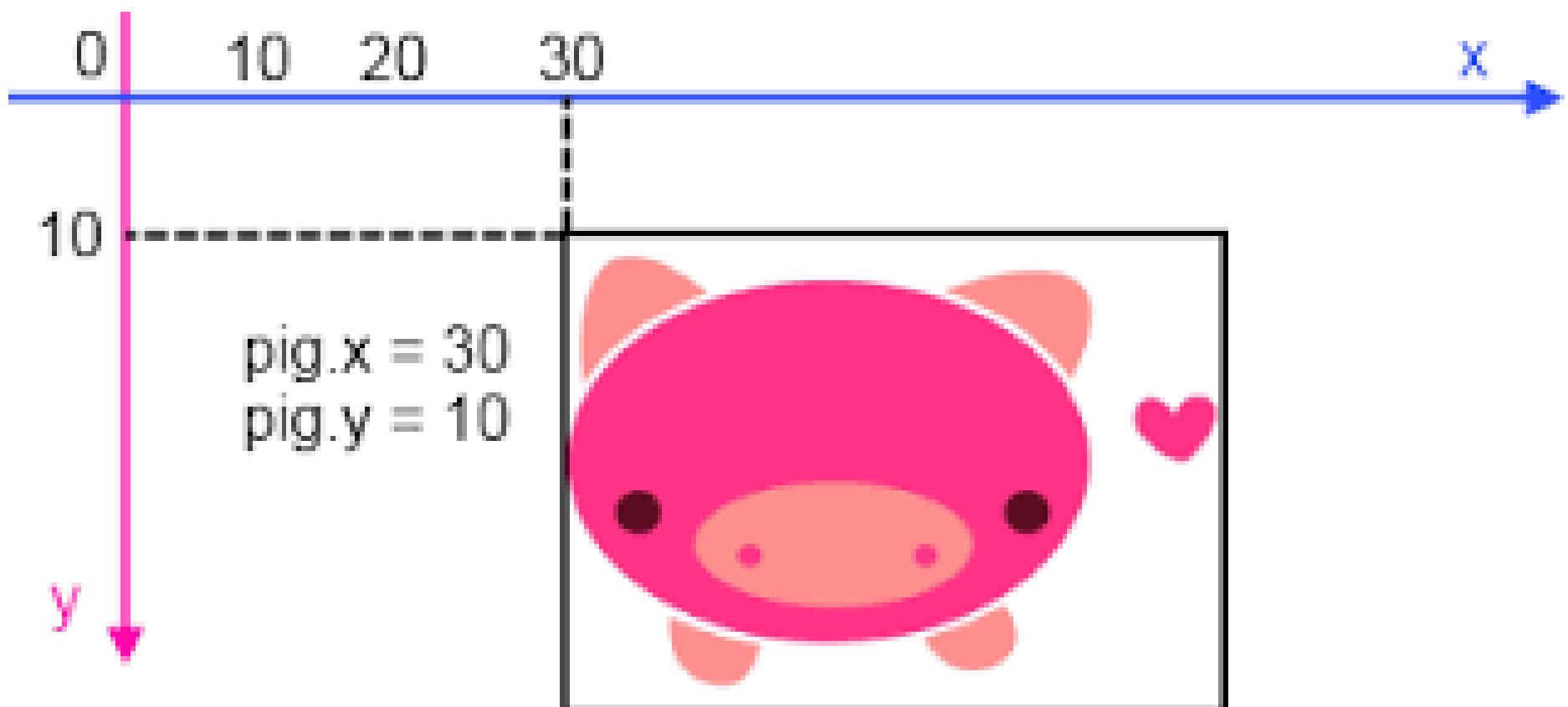


# Características

O eixo y em Löve2D é invertido! Quanto mais “pra cima” o y, mais negativo (menor) ele é! E quanto mais pra baixo, maior é o valor de y (aumenta!)

Eixo X segue normal.

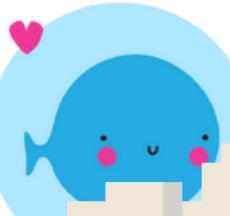
A origem de um objeto fica no seu canto superior esquerdo.



# Como criar jogos com Löve2D

**Nosso jogo será uma pasta. Nessa pasta existirão nossos arquivos .lua implementando o jogo. Para executar o nosso jogo podemos arrastar a pasta contendo esses arquivos para o arquivo love.exe.**

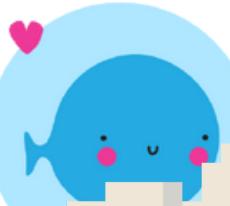
**Todo jogo em Love deve ter um arquivo chamado main.lua. Sem ele, Love não irá conseguir executar o nosso jogo.**



# Como criar jogos com Löve2D

**É bom criarmos também um arquivo chamado conf.lua (deve ter esse nome, se não não funciona) na mesma pasta para podermos configurar o nosso jogo**

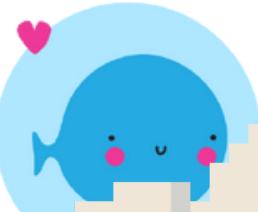
```
1 _G.love = require("love")
2
3 function love.conf(t)
4     t.title = "tutorial"
5     t.version = "11.5"
6     t.console = false
7     t.window.width = 1280
8     t.window.height = 720
9 end
```



# LUD

**No nosso arquivo main.lua devemos sempre ter o LUD:**  
**LUD é uma sigla pras funções:**  
**Load: Carrega nosso jogo**  
**Update: Atualiza a cada 60fps**  
**Draw: Desenha coisas na tela**

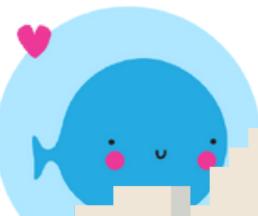
```
main.lua > ...
1  function love.load()
2  end
3
4  function love.update(dt)
5  end
6
7  function love.draw()
8  love.graphics.print("Hello World!", 400, 300)
9  end
```



# LUD

**Nós que devemos definir essas funções, por mais que elas sejam obrigatórias, pois o Love necessita que nós que definamos o que é pra acontecer com elas.**

```
main.lua > ...
1   function love.load()
2     end
3
4   function love.update(dt)
5     end
6
7   function love.draw()
8     love.graphics.print("Hello World!", 400, 300)
9   end
```

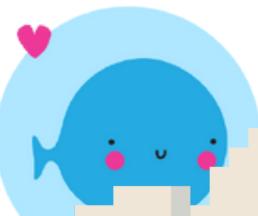


# LUD

**dt representa o tempo (em segundos) que se passou desde a última vez que love.update(dt) foi chamada.**

**Ele serve para que tudo no seu jogo funcione com a mesma velocidade, independentemente do FPS da máquina.**

```
main.lua > ...
1   function love.load()
2   end
3
4   function love.update(dt)
5   end
6
7   function love.draw()
8     love.graphics.print("Hello World!", 400, 300)
9   end
```

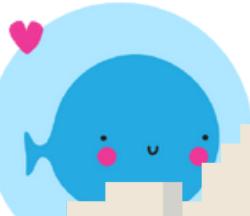


# Criando um personagem

**Para criarmos um personagem super simples podemos criar um novo arquivo, chamado como quiser, mas usaremos player.lua.**

**Nesse arquivo, iremos criar uma tabela Player = {} e também adicionaremos a definição das funções LUD, só que com a notação “ : ”, para passar uma tabela que será a instância do objeto Player.**

```
1 Player = {}
2
3 function Player:load()
4 end
5
6 function Player:update(dt)
7 end
8
9 function Player:draw()
10 end
```



# Criando um personagem

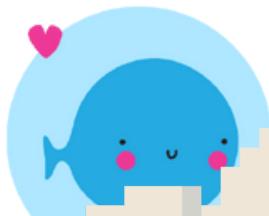
No main.lua, no início do arquivo, usaremos require("player") e passaremos as funções LUD do player em cada uma das suas respectivas funções LUD do main.lua. Isso garante que o nosso jogo:

**Load:** Carregará o player

**Update:** Atualizará o player a cada 60fps

**Draw:** Desenhe o player na tela

```
1  require("player")
2
3  function love.load()
4      Player:load()
5  end
6
7  function love.update(dt)
8      Player:update(dt)
9  end
10
11 function love.draw()
12     Player:draw()
13 end
```

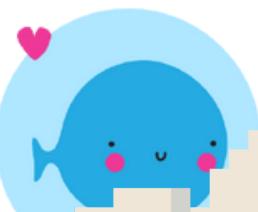


# Criando um personagem

**Agora, de volta no arquivo player.lua, iremos adicionar os atributos do player. Isso será feito na função Player:load(), já que queremos definir os atributos dele quando o jogo carregar.**

**Como chamamos o Player:load() no load do main.lua, isso vai funcionar!**

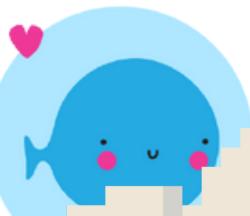
```
3 function Player:load()
4     self.x = love.graphics.getWidth() / 2
5     self.y = love.graphics.getHeight() / 2
6     self.width = 25
7     self.height = 50
8     self.speed = 300
9 end
```



# Criando um personagem

**Agora, precisamos definir o comportamento para como o player deve ser desenhado na tela:**

```
14 function Player:draw()
15     love.graphics.rectangle("fill", self.x, self.y, self.width, self.height)
16 end
```



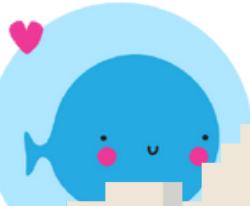
# Criando um personagem

**Agora a parte interessante! A função Player:update(dt) vai definir o que queremos que aconteça conforme o jogo atualiza, o que é basicamente o jogo em si!**

```
11 function Player:update(dt)
12     self:move(dt)
13     self:checkColisao()
14 end
```



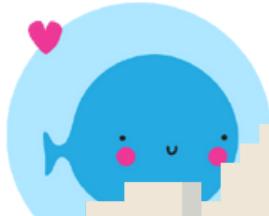
**Chamaremos dentro do update outras funções que criamos. É bom separar nosso projeto em funções.**



# Criando um personagem

## Função Player:move(dt)

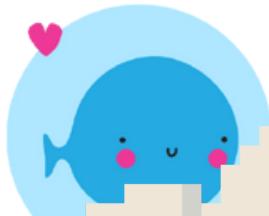
```
16     function Player:move(dt)
17         if love.keyboard.isDown("w") then
18             self.y = self.y - self.speed * dt
19         elseif love.keyboard.isDown("s") then
20             self.y = self.y + self.speed * dt
21         elseif love.keyboard.isDown("a") then
22             self.x = self.x - self.speed * dt
23         elseif love.keyboard.isDown("d") then
24             self.x = self.x + self.speed * dt
25     end
26 end
```

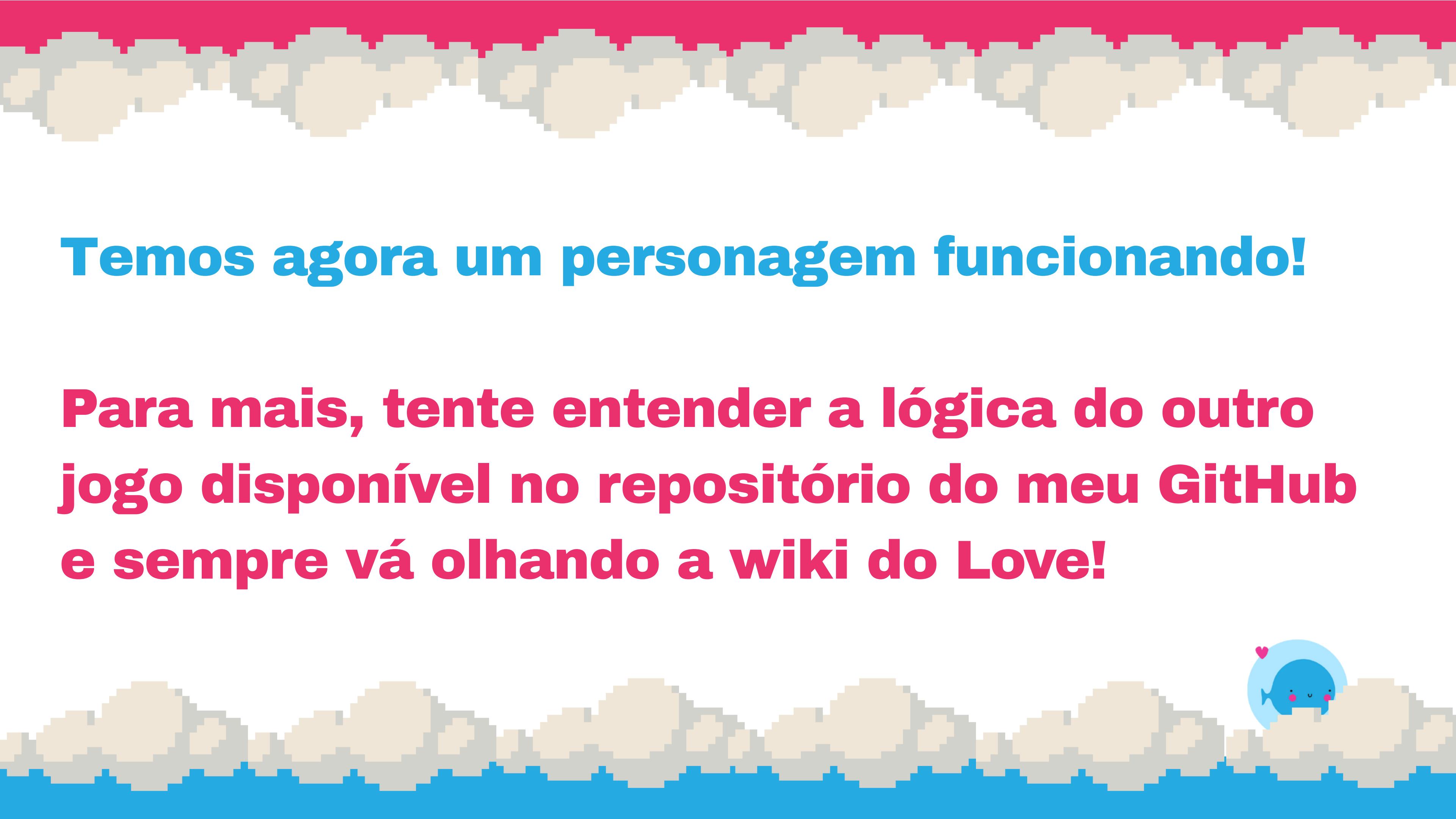


# Criando um personagem

## Função Player:checkColisao()

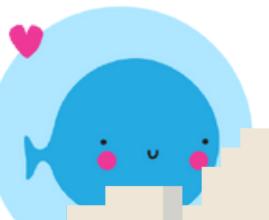
```
28     function Player:checkColisao()
29         if self.y < 0 then
30             self.y = 0
31         elseif self.y + self.height > love.graphics.getHeight() then
32             self.y = love.graphics.getHeight() - self.height
33         end
34
35         if self.x < 0 then
36             self.x = 0
37         elseif self.x + self.width > love.graphics.getWidth() then
38             self.x = love.graphics.getWidth() - self.width
39         end
40     end
```





**Temos agora um personagem funcionando!**

**Para mais, tente entender a lógica do outro  
jogo disponível no repositório do meu GitHub  
e sempre vá olhando a wiki do Love!**



# **Minicurso de Lua**

Diogo Krüger Souto

# **Fim!**

