

Lua

Linguagem de programação brasileira,
de código aberto e multiplataforma

INTEGRANTES

Alicia Monteiro, Eduardo Couto, João Vitor Fernandes,
Kleiton Josivan e Robert Danilo

SUMÁRIO

- 09** SUBPROGRAMAS
- 10** CO-ROTIÑAS
- 11** CONCORRÊNCIA ATRAVÉS DE CO-ROTIÑAS
- 12** CONCORRÊNCIA PROGRAMAÇÃO ASSÍNCRONA
- 13** TIPOS DE EXCEÇÕES E ERROS
- 14** TRATAMENTO E CRIAÇÃO DE EXCEÇÕES
- 15** DELEGAÇÃO DE RESPONSABILIDADE
- 16** TRATAMENTO DE EVENTOS EM LUA
- 17** PADRÕES COMUNS DE IMPLEMENTAÇÃO
- 18** REFERÊNCIAS

SUBPROGRAMAS: DECLARAÇÃO E CHAMADA

- ✓ Funções são valores de primeira classe
- ✓ Definição de função é uma expressão executável
- ✓ Quando executada, a função é instanciada

Modelos de passagem de parâmetros

- ➡ Passagem por valor para tipos primitivos
- ➡ Passagem por referência para tipos compostos
- ➡ Suporte a funções variádicas com “...”

Tipo do dado	Comportamento na função
number, string, boolean	Uma cópia do valor é passada. Alterações internas não afetam a variável original.
table, function, userdata	A referência é passada por valor. Alterar o conteúdo do objeto (ex: um campo da tabela) afeta o objeto original.

SUBPROGRAMAS: DECLARAÇÃO E CHAMADA

```
-- Função nomeada
function soma(a, b)
    return a + b
end

-- Chamada da função
local resultado = soma(10, 5) -- resultado será 15

-- Múltiplos retornos
function dividir(n, d)
    local quociente = math.floor(n / d)
    local resto = n % d
    return quociente, resto
end

local q, r = dividir(10, 3) -- q será 3, r será 1
```

```
-- Método (usando a sintaxe de dois pontos)
local tabela = {}
function tabela:metodo(param)
    print(self, param) -- 'self' é隐式的
end
```

```
-- Exemplo de função variádica
function exemplo(a, b, ...)
    print("Fixos:", a, b)
    print("Variáveis:", ...)
end
```

SUBPROGRAMAS: DECLARAÇÃO E CHAMADA

Closures

- Uma **closure** é uma função mais tudo o que ela precisa para acessar suas upvalues
- Upvalues** são variáveis locais externas acessadas pela função
- Permitem escopo léxico - acesso a variáveis da função envolvente

Funções Lambda

- Funções anônimas são a implementação de lambdas em Lua
- Comumente usadas como callbacks e em funções de ordem superior

Subprogramas Delegados:

- Não é um conceito formal em Lua.
- O comportamento de "delegar" uma tarefa é naturalmente implementado usando callbacks.

```
-- Exemplo de closure

function newCounter()
    local i = 0
    return function() -- função anônima (closure)
        i = i + 1 -- acessa upvalue 'i'
        return i
    end
end

c1 = newCounter()
print(c1()) --> 1
print(c1()) --> 2

-- Exemplo de função lambda como callback
nomes = {"Pedro", "Ana", "Maria"}
notas = {Maria = 10, Ana = 8, Pedro = 7}

table.sort(nomes, function(n1, n2)
    return notas[n1] > notas[n2] -- ordena por nota
end)
```

SUBPROGRAMAS: DECLARAÇÃO E CHAMADA

Programação Genérica

- Suporte através da tipagem dinâmica: Funções podem operar sobre diferentes tipos de dados.
- Tabelas como estrutura universal: Permite criar qualquer estrutura de dados genérica.
- Metatables: Permite definir comportamentos para tipos customizados.

```
-- Função genérica para encontrar um item em qualquer lista
function encontrar(lista, valor)
    for i, v in ipairs(lista) do
        if v == valor then
            return i -- Retorna o índice
        end
    end
    return nil -- Não encontrou
end

local indice_num = encontrar({10, 20, 30}, 20)      --> 2
local indice_str = encontrar({"a", "b", "c"}, "x") --> nil
```

CONCORRÊNCIA ATRAVÉS DE CO-ROTINAS

Lua nativamente trata concorrência por co-rotinas, mas há bibliotecas como Lanes e LuaProcess para threads reais.

- ⟳ Implementam multitarefa cooperativa em um único thread
- ⟳ Podem **suspender** sua execução e **retomar** posteriormente
- ⟳ Possuem três estados: **suspended**, **running** e **dead**

Funções principais

- </> **coroutine.create(f)**: cria uma nova co-rotina
- </> **coroutine.resume(co, ...)**: inicia/retoma a execução
- </> **coroutine.yield(...)**: suspende a execução

```
local co = coroutine.create(function()
    print("Co-rotina iniciada")
    coroutine.yield(1)
    print("Co-rotina retomada")
end)

print(coroutine.status(co)) -- suspended
coroutine.resume(co)        -- Imprime "Co-rotina iniciada" e retorna 1
print(coroutine.status(co)) -- suspended
coroutine.resume(co)        -- Imprime "Co-rotina retomada"
print(coroutine.status(co)) -- dead
```

TIPOS DE EXCEÇÕES E ERROS

Lua não tem uma hierarquia complexa de exceções. Um "erro" é qualquer condição que interrompe o fluxo normal.

- **Erros de Sintaxe:** Detectados durante a compilação
- **Erros de Tempo de Execução:** Acesso a índice nulo, operações com tipos incompatíveis
- **Erros Personalizados:** Criados pelo programador com a função error()

```
-- Exemplo de erro de tempo de execução  
local function acessarIndice(tabela, indice)  
    return tabela[indice] -- Pode gerar erro se tabela for nil  
end
```

```
-- Exemplo de erro personalizado  
function dividir(a, b)  
    if b == 0 then  
        error("Divisão por zero não permitida")  
    end  
    return a / b  
end
```

```
-- Exemplo de erro com valor não-string  
function validarIdade(idade)  
    if idade < 0 then  
        error({codigo = 101, mensagem = "Idade inválida"})  
    end  
    return true  
end
```

TRATAMENTO E CRIAÇÃO DE EXCEÇÕES

Capturando Erros: Feito com a função `pcall` (Protected Call).

```
function pode_falhar()
    -- Tenta uma operação inválida
    return "a" + 1
end

local sucesso, resultado = pcall(pode_falhar)

if sucesso then
    print("Resultado:", resultado)
else
    print("Ocorreu um erro:", resultado) -- Imprime a mensagem de erro
end
```

- **xpcall:** Uma versão de `pcall` que permite especificar uma função de callback para tratar o erro.
- Criação de "Exceções" Customizadas: Não se cria uma classe de exceção. Em vez disso, lança-se um erro com uma tabela, permitindo anexar mais informações.
 1. `error(mensagem, [nível]):` Lança um erro e termina a execução (se não for capturado).
 2. `assert(condição, [mensagem]):` Se a condição for `false` ou `nil`, lança um erro.

```
error({ code = 404, message = "Recurso não encontrado" })
```

DELEGAÇÃO DE RESPONSABILIDADE

- Se um erro não é capturado por pcall, ele "sobe" na pilha de chamadas.
- Delegação Implícita: Uma função pode optar por não tratar um erro, delegando essa responsabilidade para a função que a chamou.

```
function nivel_inferior()
    error("Problema aqui em baixo!") -- Lança o erro
end

function nivel_superior()
    -- Não trata o erro, ele se propaga
    nivel_inferior()
end

-- O tratamento é feito no nível mais alto da aplicação
local sucesso, erro = pcall(nivel_superior)
if not sucesso then
    print("Erro capturado:", erro)
end
```

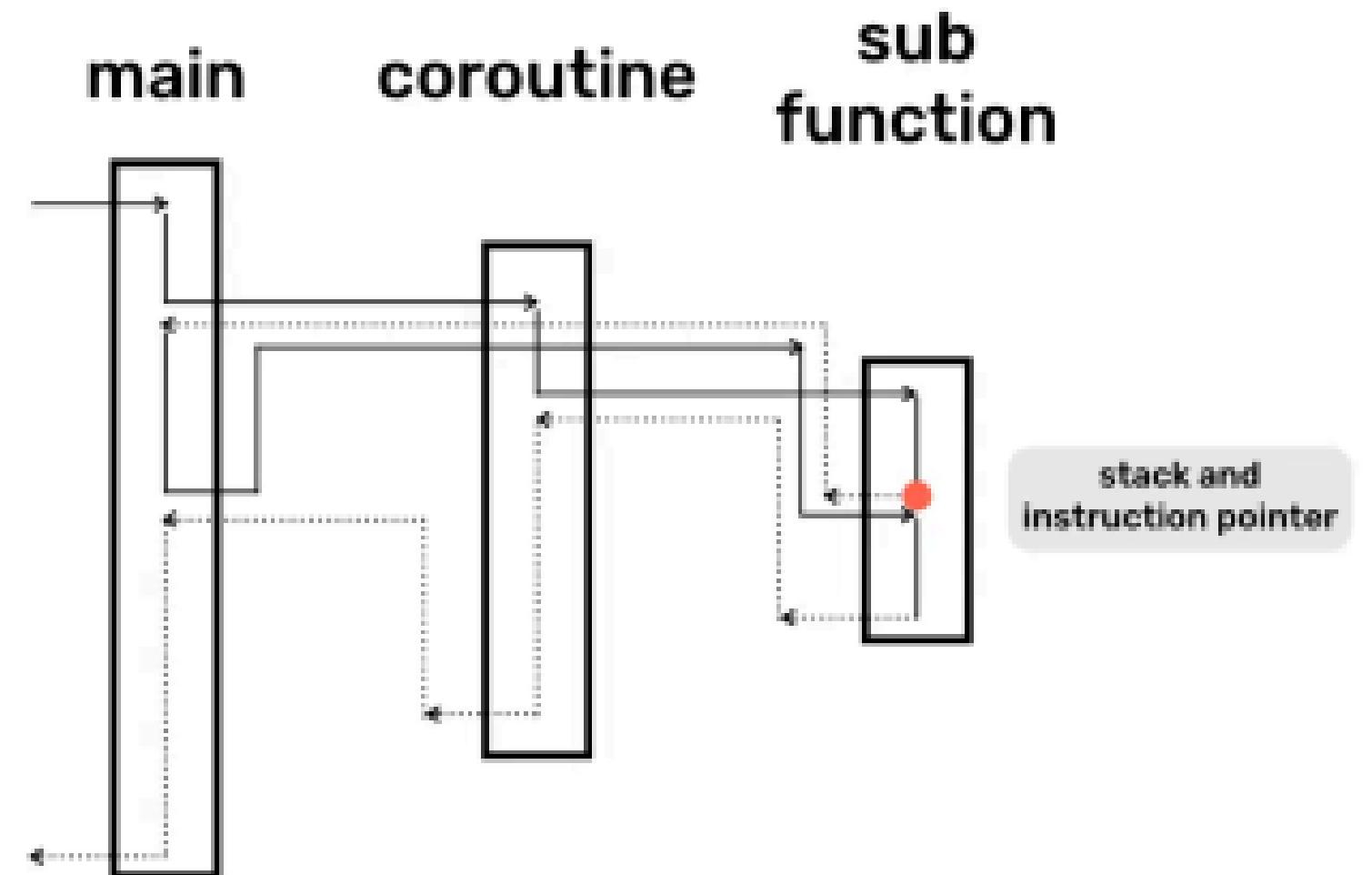
TRATAMENTO DE EVENTOS EM LUA

- Não é uma característica da linguagem: A biblioteca padrão de Lua não possui um sistema de eventos (event loop).
- Dependente da aplicação hospedeira: O tratamento de eventos é uma função do ambiente onde Lua está embarcada.

Ambiente	Exemplo de Eventos	Modelo de Implementação
Game Engine (ex: LÖVE)	love.draw, love.keypressed	Funções de callback globais.
UI Toolkit (ex: IUP)	Clique de botão, movimento do mouse	Associa funções Lua a callbacks de widgets.
Servidor Web (ex: OpenResty)	Requisição HTTP recebida	Scripts Lua são executados em fases do ciclo de vida da requisição.

CONCORRÊNCIA PROGRAMAÇÃO ASSÍNCRONA

- **Paralelismo real:** A biblioteca padrão de Lua não oferece suporte a threads de sistema operacional para execução paralela em múltiplos núcleos de CPU.
- **Programação Assíncrona:** É o ponto forte de Lua. Um scheduler (geralmente provido por um framework ou pela aplicação hospedeira) gerencia múltiplas co-rotinas. Quando uma co-rotina precisa esperar por uma operação de I/O, ela usa yield, e o scheduler passa a execução para outra, evitando o bloqueio.
- **Bibliotecas externas:** Para paralelismo, projetos como lanes e lua-llthreads podem ser usados para integrar Lua com threads do SO.



PADRÕES COMUNS DE IMPLEMENTAÇÃO

- **Funções de Callback:** O método mais comum. Você registra uma função Lua para ser chamada quando um evento acontece.

```
-- Em um framework hipotético
botao:setOnClick(function()
    print("Botão clicado!")
end)
```

- **Metatables:** Podem ser usadas para simular eventos em tabelas. Por exemplo, o metamétodo `_newindex` pode ser usado para disparar uma ação sempre que um valor em uma tabela é alterado.
- **Co-rotinas:** Em sistemas avançados, um evento externo pode resumir uma co-rotina que estava em `yield`, esperando por aquele evento.

CONCLUSÃO

- **Subprogramas:** Funções são "cidadãos de primeira classe", permitindo padrões avançados como closures e múltiplos retornos.
- **Concorrência:** Baseada em multitarefa cooperativa (co-rotinas) para controle explícito de tarefas assíncronas, sem paralelismo nativo.
- **Tratamento de Exceções:** Modelo seguro e pragmático centrado na função pcall para capturar erros sem travar a aplicação.
- **Tratamento de Eventos:** Atua como linguagem de extensão, respondendo a eventos da aplicação hospedeira através de callbacks.
- **Paradigmas e Metaprogramação:** Suporta múltiplos paradigmas (OO, funcional) e permite metaprogramação (modificar a própria linguagem) via metatables.

REFERÊNCIAS

- Documentação oficial: <http://www.lua.org/docs.html>
- Livro online gratuito (Lua 5.0): <http://www.lua.org/pil/contents.html>
- Repositório GitHub: <https://github.com/aliciamonteiro/Lua>

OBRIGADO

pela atenção!

INTEGRANTES

Alicia Monteiro, Eduardo Couto, João Vitor Fernandes,
Kleiton Josivan e Robert Danilo

alicia20230028449@alu.uern.br
eduardo20230028743@alu.uern.br
joao20230021214@alu.uern.br
kleiton20230036226@alu.uern.br
robert20230023200@alu.uern.br

ALGUMA DÚVIDA?