+ Lua + Löve

callbacks e estado

```
function love.keypressed(key)
 local mx, my = love.mouse.getPosition()
  if key == 'b' and naimagem (mx,my, x, y) then
     y = 200
  end
end
function love.update (dt)
  local mx, my = love.mouse.getPosition()
  if love.keyboard.isDown("down") and naimagem(mx, my, x, y) then
    y = y + 10
  end
end
function love.draw ()
  love.graphics.rectangle("line", x, y, w, h)
end
```

- variáveis globais capturam estado como no Arduino
- mas podemos usar características da linguagem para encapsular esse estado

encapsulando estado

```
function retangulo (x,y,w,h)
  local originalx, originaly, rx, ry, rw, rh =
                                  x, y, x, y, w, h
  return {
    draw =
      function ()
        love.graphics.rectangle("line", rx, ry, rw,
rh)
      end,
    keypressed =
      function (key)
        local mx, my = love.mouse.getPosition()
        ...
      end
  }
end
function love.load()
  ret1 = retangulo (50, 200, 200, 150);
end
```

encapsulando estado

```
function love.load()
  ret1 = retangulo (50, 200, 200, 150);
end

function love.keypressed(key)
  ret1.keypressed(key)
end
```

onde colocar a fç `na imagem`?

encapsulando estado

```
function retangulo (x,y,w,h)
  local originalx, originaly, rx, ry, rw, rh =
                                    x, y, x, y, w, h
  return {
    draw =
      function ()
        love.graphics.rectangle("line", rx, ry, rw,
rh)
      end,
    keypressed =
      function (key)
        local mx, my = love.mouse.getPosition()
        ...
      end
  }
end
function naimagem (???) (
  -- w e h globais (?) ou passadas como args
  return (mx>x) and (mx<x+w) and (my>y) and (my<y+h)
end
```

encapsulando estado

```
function retangulo (x,y,w,h)
  local originalx, originaly, rx, ry, rw, rh =
                                    x, y, x, y, w, h
  return {
    naimagem = function (mx, my)
      end
    draw =
      function ()
        love.graphics.rectangle("line", rx, ry, rw,
rh)
      end,
    keypressed =
      function (self, key)
        local mx, my = love.mouse.getPosition()
        if self.naimagem(mx, my) then ...
        ...
      end
  }
end
```

encapsulando estado

```
function retangulo (x,y,w,h)
  local originalx, originaly, rx, ry, rw, rh =
                                  x, y, x, y, w, h
  local function naimagem (mx, my)
    return
     (mx>rx) and (mx<rx+rw) and (my>ry) and (my<ry+rh)
  end
  return {
    draw =
      function ()
        love.graphics.rectangle("line", rx, ry, rw, rh)
      end,
    keypressed =
      function (key)
        local mx, my = love.mouse.getPosition()
        ...
      end
  }
end
```

mais Lua: co-rotinas

- semelhante a uma thread
  - pilha, variáveis locais e instruction pointer (PC) próprios
- execução *colaborativa*
  - apenas uma co-rotina executa a cada momento
- funções definidas na tabela `coroutine`

co-rotina: ex 1

```
co = coroutine.create ( function ()
       for i = 1, 4 do
         print ("co", i)
         coroutine.yield()
       end
      end)
print(coroutine.type(co)) -> thread
coroutine.resume(co) -> co 1
                        true
coroutine.resume(co) -> co 2
coroutine.resume(co) -> co 3
coroutine.resume(co) -> co 4
coroutine.resume(co) … nada
coroutine.resume(co) -> cannot resume dead coroutine
```

co-rotinas: troca de valores

- yield e resume podem trocar valores

```
co = coroutine.create ( function ()
       for i = 1, 10 do
         print("dentro recebeu: "..
                coroutine.yield(i))
       end
     end)

for j = 11, 20 do
  _, ret = coroutine.resume (co, j)
  print ("fora recebeu: " .. ret)
end
```

```
fora recebeu:  1
dentro recebeu:  12
fora recebeu:  2
dentro recebeu:  13
fora recebeu:  3
dentro recebeu:  14
fora recebeu:  4
dentro recebeu:  15
fora recebeu:  5
dentro recebeu:  16
fora recebeu:  6
dentro recebeu:  17
fora recebeu:  7
dentro recebeu:  18
fora recebeu:  8
dentro recebeu:  19
fora recebeu:  9
dentro recebeu:  20
fora recebeu:  10
```

co-rotinas: troca de valores

- yield e resume podem trocar valores

```
co = coroutine.create ( function ()
       for i = 1, 10 do
         print("dentro: "..
               coroutine.yield(i))
       end
     end)

for j = 11, 20 do
  _, ret = coroutine.resume (co, j)
  print ("fora: " .. ret)
end
```

- *onde foi parar o valor 11...?*

```
fora: 1
dentro: 12
fora: 2
dentro: 13
fora: 3
dentro: 14
fora: 4
dentro: 15
fora: 5
dentro: 16
fora: 6
dentro: 17
fora: 7
dentro: 18
fora: 8
dentro: 19
fora: 9
dentro: 20
fora: 10
```

co-rotinas: troca de valores

```
co = coroutine.create ( function ()
       for i = 1, 10 do
         print("dentro: "..
                coroutine.yield(i))
       end
     end)

repeat
  local status, ret = coroutine.resume (co, j)
  if status then print ("fora: " .. ret)
until not status
```

co-rotinas: troca de valores

- wrap: fç auxiliar conveniente

```
co = coroutine.wrap ( function ()
       for i = 1, 10 do
         print("dentro: "..
                coroutine.yield(i))
       end
     end)

for j = 11, 20 do
  print ("fora: " .. co(j))
end
```

```
fora:  1
dentro:  12
fora:  2
dentro:  13
fora:  3
dentro:  14
fora:  4
dentro:  15
fora:  5
dentro:  16
fora:  6
dentro:  17
fora:  7
dentro:  18
fora:  8
dentro:  19
fora:  9
dentro:  20
fora:  10
```

co-rotinas: iteradores

exemplo: permutações

```
function permgen (a, n)
  n = n or #a
  if n <= 1 then
    printResult(a)
  else
    for i=1,n do
      -- put i-th element as the last one
      a[n], a[i] = a[i], a[n]
      -- generate all permutations of the other elements
      permgen(a, n - 1)
      -- restore i-th element
      a[n], a[i] = a[i], a[n]
    end
  end
end
```

do livro *Programming in Lua*

exemplo: permutações

```
function permgen (a, n)
  n = n or #a
  if n <= 1 then
    coroutine.yield(a) — retorna uma sequência!!!
  else
    for i=1,n do
      -- put i-th element as the last one
      a[n], a[i] = a[i], a[n]
      -- generate all permutations of the other elements
      permgen(a, n - 1)
      -- restore i-th element
      a[n], a[i] = a[i], a[n]
    end
  end
end
```

do livro *Programming in Lua*

exemplo: permutações

```
function permgen(a, n)
  n = n or #a
  if n <= 1 then
    coroutine.yield(a)
  else
    for i=1,n do
      a[n], a[i] = a[i], a[n]
      permgen(a, n - 1)
      a[n], a[i] = a[i], a[n]
    end
  end
end
```

```
function permutations (a)
  local co = coroutine.create (function () permgen (a) end)
  return function () -- iterator
    local code, res = coroutine.resume(co)
    return res
  end
end

for p in permutations{"a", "b", "c"} do
  ...
end
```

do livro *Programming in Lua*

co-rotinas: multithreading

```
function create_task(f)  -- cria uma tarefa
  local co = coroutine.create(f)
  table.insert(tasks, co)
end
function dispatcher()  -- escalonador de tarefas
 ...
end
```

co-rotinas: multithreading

```
function dispatcher()  -- escalonador de tarefas
  local i = 1
  while true do
    if tasks[i] == nil then
      if tasks[1] == nil then break end
      i= 1
    end
    local status = coroutine.resume(tasks[i])
    if status==false then
      print ("acabou uma tarefa")
      table.remove(tasks, i)
    else
      i= i + 1
    end
  end
  print ("acabaram as tarefas")
end
```

uso em löve

- cada "personagem" (jogador ou não) pode ter seu comportamento descrito por uma co-rotina
  - estado armazenado em pilha
  - no nosso exercício anterior, poderíamos usar resume em vez de nova invocação de `rect[i].draw`

exercício 1

- código em

  ~noemi/sr-19/code/lovelua/circulos/

  transformar função que desenha em co-rotina com chamadas recursivas.

  antes da chamada recursiva colocar o yield

exercício 2

- código em

  ~noemi/sr-19/code/lovelua/retangulo2/

exercício 2

- criar uma chamada wait(segundos, meublip), e trocar chamada a coroutine.yield por chamada a essa chamada
  - wait deve ser não bloqueante: a fç deve ”marcar” que o blip que a chamou está inativo e em seguida dar coroutine.yield
  - a fç love.update agora deve manter a hora corrente e, ao percorrer os blips, chamar update apenas para os ativos e verificar se está na hora de reativar algum blip inativo
  - o parâmetro vel não deve mais ser usado como incremento de x, e sim para determinar o tempo de espera em inatividade
    - todos os blips devem atualizar x com o mesmo valor

exercicio

```
update = function (self)
  local width, height = love.graphics.getDimensions()
  x = x+vel
    if x > width then
    -- volta à esquerda da janela
      x = 0
    end
  end,

 update = coroutine.wrap ( function ()
   while true do
     local width, height = love.graphics.getDimensions()
     x = x + sec
     if x > width then x = 0 end
     coroutine.yield()
   end
end) ,
```

exercicio

```
update = function (self)
  local width, height = love.graphics.getDimensions()
  x = x+vel
    if x > width then
    -- volta à esquerda da janela
      x = 0
    end
  end,

 update = coroutine.wrap ( function (self)
   while true do
     local width, height = love.graphics.getDimensions()
     x = x + sec
     if x > width then x = 0 end
     wait(sec/100,self)
   end
end) ,
```

update como scheduler

- a fç love.update está fazendo o papel de um escalonador
  - blips ativos seriam nossas tarefas ”prontas”
  - update só irá chamar os blips ativos
    - criar fç wait e possivelmente outras auxiliares

miniprojeto löve

- usar estrutura anterior para desenvolver um jogo mais sofisticado
- entrega em 12/5
  - zip de arquivos pelo ead
- apresentação em 14/5
  - *apresentação!*