metatables

the one mechanism lua gives you to change what an operation means

what youll have by the end

you can read any `__index` you meet, write a class with methods and inheritance, make your own type do `+`, `==`, `#` and `print`, and say exactly what lua is doing at each step

no prior lua needed, we build it from a bare table up

the big picture

lua has almost no built in structure, no classes, no objects, no interfaces

it has **tables**, one container that does the job of a list, a dictionary and an object, and thats it

a metatable is the hook that lets you say **what should happen when someone does X to this table**, and thats the whole feature

a plain table has no opinions

```lua
local t = { name = "finn" }

print(t.name)    --> finn
print(t.age)     --> nil
print(t + 1)     --> error, attempt to perform arithmetic on a table value
```

ask for a key it doesnt have, you get `nil`, quietly

try to add it to a number, lua gives up and errors

thats the default behaviour, and a metatable is how we replace it

like a sticky note on the filing cabinet

a table is a filing cabinet, you open a drawer by name and take whats in it

a metatable is a **sticky note stuck on the lid**, and the note says things like

"drawer empty ? go ask the cabinet next door", or *"someone tried to add me to something ? here's how"*

lua reads that note only when it gets stuck, never when things just work

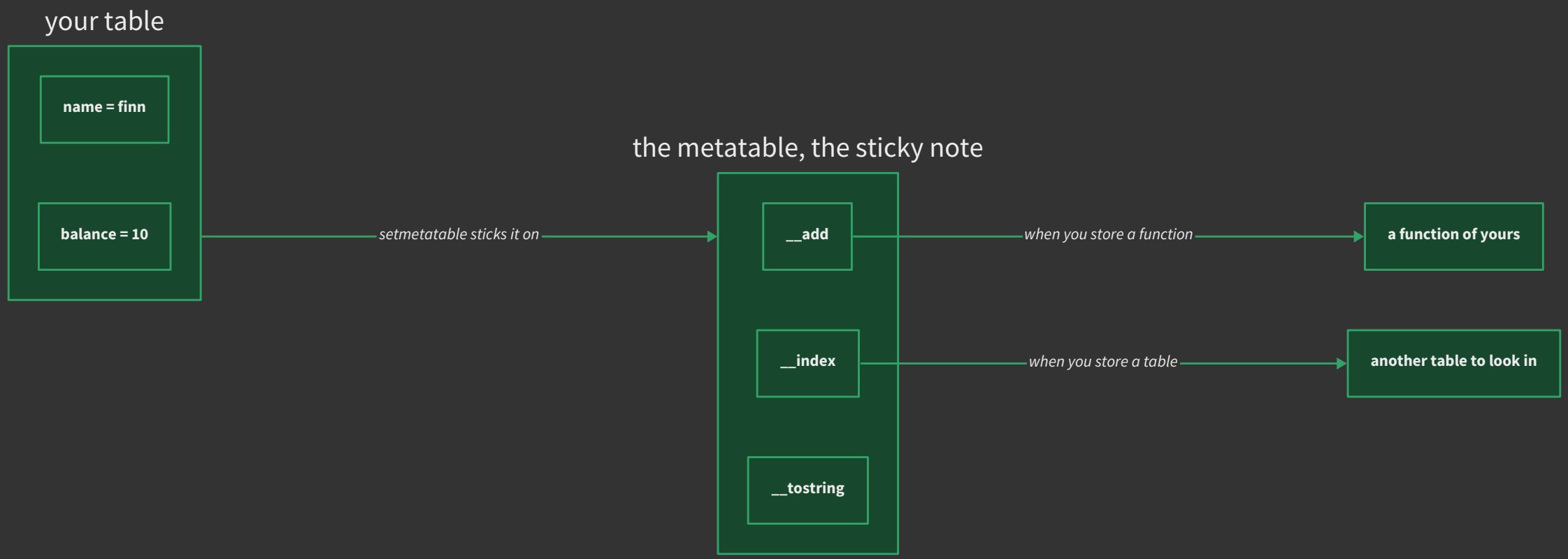
two words before we start

a raw operation means the plain one, no sticky note consulted, just look in the drawer and report what you found

a metamethod means one entry on the note, a value stored under a key like `__index` or `__add`, the double underscore is just naming, lua looks those exact names up

every metamethod is either a function lua calls, or a table lua looks in, and which one you use changes the meaning a bit

the map


your table
name = finn
balance = 10
setmetatable sticks it on
the metatable, the sticky note
__add
when you store a function
a function of yours
__index
when you store a table
another table to look in
__tostring

attaching one

```lua
local mt = {}
local t  = setmetatable({}, mt)

print(getmetatable(t) == mt)  --> true
```

`setmetatable(tbl, mt)` sticks the note on, and **returns the table back**, so you can build and attach in one expression, thats why its so often written inline

`getmetatable(tbl)` peels it back off, `nil` when theres none

the rule that governs everything

> a metamethod only fires when the normal thing **fails**

thats the sentence to keep, every confusing metatable bug comes back to it

`t.name` when `name` is there, the note is never read

`t.age` when `age` is missing, now lua reads the note

so a metatable is a **fallback**, not an interceptor, by default

__index, as a table

the most used one by a mile, missing key, go look over there instead

```lua
local defaults = { colour = "blue", size = 10 }
local cfg      = setmetatable({ size = 42 }, { __index = defaults })

print(cfg.size)    --> 42, its right there, no note read
print(cfg.colour)  --> blue, missed, so lua went to defaults
print(cfg.shape)   --> nil, missed in both, so nil
```

`cfg` still only holds `size`, nothing got copied, the lookup just carried on somewhere else

__index, as a function

same slot, but hand it a function and lua **calls** it with the table and the key that missed

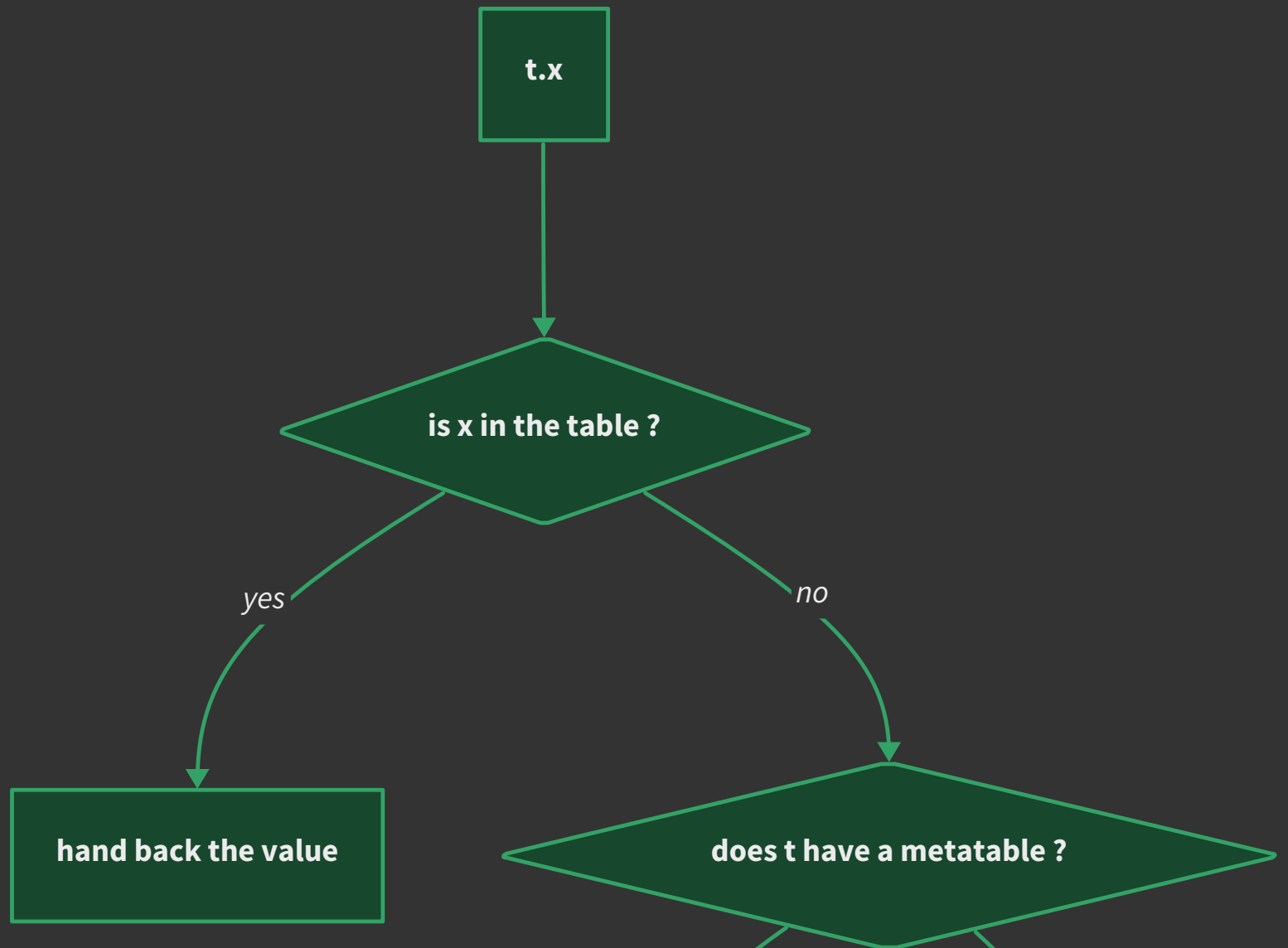
```lua
local t = setmetatable({}, {
        __index = function(tbl, key)
                return "no idea what " .. key .. " is"
        end
})

print(t.anything)  --> no idea what anything is
```

whatever you return becomes the value of `t.anything`

a table for a plain fallback, a function when you want to compute it, log it, or error on purpose

so what lua actually does on t.x



the chain falls out for free

`__index` pointing at a table means that table gets the **whole lookup** again, metatable and all

```lua
local animal = { legs = 4 }
local dog    = setmetatable({ bark = true }, { __index = animal })
local rex    = setmetatable({ name = "rex" }, { __index = dog })

print(rex.legs)  --> 4
```

`rex` misses, goes to `dog`, misses, goes to `animal`, finds it

thats inheritance, and nobody had to build it, it is just the same rule applied twice

__newindex, the write side

the mirror of `__index`, it fires when you assign to a key the table **does not already have**

```lua
local t = setmetatable({}, {
        __newindex = function(tbl, key, val)
                print("someone set " .. key .. " to " .. tostring(val))
        end
})

t.hp = 100        --> someone set hp to 100
print(t.hp)       --> nil
```

look at that second line, our function printed and returned, it never stored anything, so the table is still empty

which is the classic trap

if you take over `__newindex` you are now responsible for the storing

```
__newindex = function(tbl, key, val)
        print("set " .. key)
        rawset(tbl, key, val)   -- this is the bit people forget
end
```

`rawset` writes straight into the table with **no note consulted**, so we dont loop back into ourselves forever

and note the same once only catch, `t.hp = 100` fires the note, `t.hp = 200` right after does not, bc `hp` exists now

the raw family, your escape hatches

four functions that do the plain operation and ignore the metatable completely

rawget(t, k) look in the drawer, no fallback, this is how you ask *does it really have this*

rawset(t, k, v) store it, no `__newindex`

rawequal(a, b) compare identity, no `__eq`

rawlen(t) the real length, no `__len`

you reach for these inside a metamethod, bc thats exactly where calling the normal version would recurse into you again

the operator metamethods

every operator in the language has one, the names map straight onto the symbol

```
__add  +      __sub  -      __mul  *      __div  /
__mod  %      __pow  ^      __unm  -x     __idiv //
__band &      __bor  |      __bxor ~      __bnot ~x
__shl  <<     __shr  >>     __concat ..   __len  #
__eq   ==     __lt   <      __le   <=     __call t()
```

theyre all the same shape, lua hits an operator it cant do on your type, reads the note, calls what it finds

so learn one and you have learnt all of them

__add, worked through

```lua
local vec = {}
vec.__index = vec

vec.__add = function(a, b)
        return setmetatable({ x = a.x + b.x, y = a.y + b.y }, vec)
end

local p = setmetatable({ x = 1, y = 2 }, vec)
local q = setmetatable({ x = 3, y = 4 }, vec)
local r = p + q

print(r.x, r.y)  --> 4 6
```

lua sees `p + q`, neither is a number, so it looks for `__add` and calls it with both operands, whatever you return is the result

whose metamethod gets used

`p + 5`, one side is a table and one is a number, so which note does lua read

it tries the **first** operand, and if that has no `__add`, it tries the **second**

```
local n = 5 + p   -- still calls vec.__add, found on the second operand
```

which means your function has to cope with its arguments arriving either way round, a `+` is not guaranteed to hand you your own type in slot one

comparison, and the catch on __eq

```
vec.__eq = function(a, b) return a.x == b.x and a.y == b.y end
vec.__lt = function(a, b) return a.x <  b.x end
vec.__le = function(a, b) return a.x <= b.x end
```

`__eq` is fussier than the rest, lua only calls it when both sides are the **same kind of thing**, two tables or two userdata, and theyre not already the identical object

so `p == 5` is just `false`, your function never runs, and `p == p` is `true` without running it either

`~=` is `not __eq`, and `>` and `>=` are `<` and `<=` with the operands flipped, you never write those four

the small ones

__concat fires on `..` when either side is not a string or a number, same first then second operand rule

__len fires on `#`, return whatever you want the length to be, handy for a table that stores its count in a field

__unm fires on a negation, `-p`, it still gets called with two arguments and both are your value, just use the first

__call, a table that behaves like a function

```lua
local counter = setmetatable({ n = 0 }, {
        __call = function(self, step)
                 self.n = self.n + (step or 1)
                 return self.n
        end
})

print(counter())   --> 1
print(counter(10)) --> 11
```

`counter(10)` becomes `__call(counter, 10)`, the table slides in as the first argument

so you get a thing that calls like a function but still has fields you can read, which a plain function cannot do

__tostring and __name

```lua
vec.__tostring = function(v) return "vec(" .. v.x .. ", " .. v.y .. ")" end
vec.__name     = "vec"

print(p)  --> vec(1, 2)
```

`__tostring` is what `print` and `tostring` call instead of printing `table: 0x600001c...`

`__name` is a plain string, no function, lua uses it in **error messages** and as the fallback label when theres no `__tostring`, so an error reads `bad argument, vec expected` rather than `userdata expected`

worth setting both, theyre free and they make debugging far less miserable

youve been using one already

```lua
local s = "hello"
print(s:upper())      --> HELLO
print(("%d"):format(7))  --> 7
```

strings are not tables, so how does `s:upper()` find anything

every string in lua shares one metatable, and it has `__index = string`, the standard string library

so `s:upper` misses on the string, lua reads the note, looks in `string`, finds `string.upper`, done, exactly the mechanism from four slides ago

numbers and booleans have **no** metatable at all by default, which is why `(5):abs()` errors

locking it down, __metatable

```lua
local t = setmetatable({}, { __metatable = "private" })

print(getmetatable(t))      --> private
setmetatable(t, {})         --> error, cannot change a protected metatable
```

set the `__metatable` field to anything and `getmetatable` hands that back instead of the real note, while `setmetatable` refuses outright

useful when youre handing a table to code you dont trust and you dont want it swapping your rules out

__mode and __pairs

__mode makes the table weak, set it to `"k"`, `"v"` or `"kv"` and entries with only weak references left get collected rather than kept alive, thats how you write a cache that doesnt leak, set it before the table is in use

__pairs takes over `pairs(t)`, return the iterator, the table and the start value and your loop runs through whatever you decide to expose

both are niche, but they show the range, the note reaches the garbage collector and the loop constructs too

__gc and __close, the lifetime pair

__gc runs when the value gets collected, your chance to close a file or free a handle, the catch is the metatable must already have `__gc` in it **at the moment you call setmetatable**, added afterwards it is ignored

__close runs when a to be closed variable goes out of scope, deterministically, no waiting on the collector

```
do
        local f <close> = open_thing()
        -- __close runs right here, even on an error
end
```

`__gc` is best effort and whenever, `__close` is exact and now, reach for `__close` when the order matters

the colon, before we do classes

```
obj:method(1)   -- is exactly
obj.method(obj, 1)
```

thats all the colon is, sugar that passes the thing on the left as the first argument

and by convention that first argument is named `self`, so `function t:m(a)` and `function t.m(self, a)` are the same function written two ways

every lua class trick is built on this plus `__index`, nothing else

a class, the whole pattern

```lua
local account = {}
account.__index = account

function account.new(owner, balance)
        return setmetatable({ owner = owner, balance = balance or 0 }, account)
end

function account:deposit(amount)
        self.balance = self.balance + amount
end
```

that middle line is the trick, **the class is its own metatable**

why that one line works

`account.__index = account` means, when a lookup misses on an instance, go look in `account`

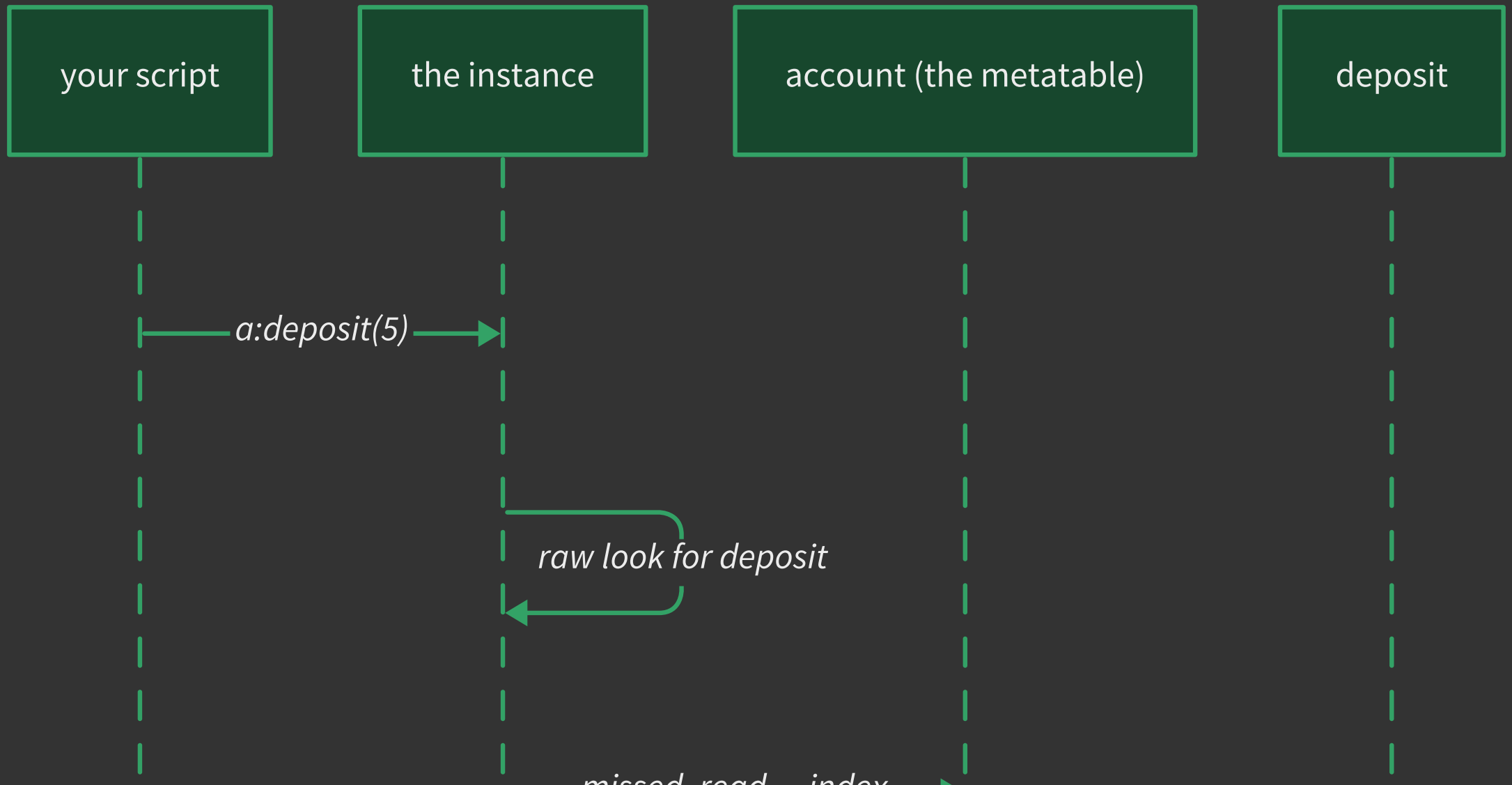
```lua
local a = account.new("finn", 10)
a:deposit(5)
```

`a` holds two fields, `owner` and `balance`, it does **not** hold `deposit`

`a:deposit` misses, lua reads the note, `__index` points at `account`, `deposit` is right there, lua calls it with `a` as `self`

one table holds the methods for every instance ever made, thats the saving

a method call, end to end



inheritance, same trick twice

```lua
local savings = setmetatable({}, { __index = account })
savings.__index = savings

function savings:add_interest(rate)
        self.balance = self.balance * (1 + rate)
end
```

read those two lines separately, they do different jobs

line one gives `savings` itself a note pointing at `account`, so the class inherits

line two gives instances of `savings` a note pointing at `savings`, so objects find their methods

miss on the object, go to `savings`, miss again, go to `account`, the chain from earlier doing all the work

what metatables do not do

theyre per table, not per field, one note covers the whole thing, you cannot guard a single key without guarding them all

nil and false are not the same miss, a key stored as `false` is present, so `__index` never fires for it, only a genuinely absent key does

a raw call skips everything, so `rawget` inside your own code will not see inherited fields, thats the point of it but it catches people

sharing a metatable shares the rules, hand the same note to a hundred tables and they all behave identically, which is usually what you want and occasionally a surprise

from the c side

everything here has a c api twin, and thats where it really earns its keep

`luaL_newmetatable(L, "vec")` makes a named metatable in the registry, `luaL_setmetatable` sticks it on a userdata, `__gc` frees the c++ object behind it, and `__index` pointing back at the metatable gives your c type methods

if you want that end, the two api lectures cover it in full

- `lectures/lua-c-api-writing-a-library.slides.md`, part 1, the raw api
- `lectures/lua-c-api-binding-lib.slides.md`, part 2, generating the boilerplate

why its built this way

- **one mechanism, not ten**, lua has no class keyword, no operator overloading syntax, no property system, it has the note and every one of those is written in terms of it
- **fallback, not interception**, checking the note only after the plain path failed means a normal hit costs nothing, the fast case stays fast
- **tables all the way down**, a metatable is an ordinary table, so you can inspect it, share it, build it at runtime, pass it around
- **the cost is honesty**, lua gives you the mechanism and no policy, so every codebase invents its own class pattern, and you have to read which one youre in

recap

- a metatable is a table of rules stuck to another value, and lua reads it **only when the plain operation fails**
- `__index` handles missing reads, a table to redirect, a function to compute, and it recurses, which is where inheritance comes from
- `__newindex` handles new writes, and if you take it over you must `rawset` the value yourself
- every operator has a metamethod, lua checks the first operand then the second
- a class is just `Class.__index = Class` plus `setmetatable` in the constructor, and the colon passing `self`
- the raw family skips the whole system, use it inside metamethods to stop the recursion

have a go

write a **read only** table

```
local function freeze(t)
        -- __index hands back the real values
        -- __newindex errors instead of storing
        -- the data has to live somewhere the user cannot reach
end
```

two hints, if you keep the data in the table itself then `__index` never fires bc the keys are present, so keep it in a hidden table the note closes over

and check your error actually stops the write, `rawset` would go straight past you