
Lua

Booklet 2 of 2: Metatables to embedding

Chapters 5 to 8

Fall 2026

Contents

Chapter 5: Metatables and OOP.....	1
Tables as Sets	1
Metatables Basics	5
Object-Oriented Programming with Metatables	11
Chapter 6: Standard Library and Modules.....	17
Introduction to the Standard Library.....	17
Using Modules in Lua	23
Chapter 7: Coroutines.....	29
Introduction to Coroutines	29
Using Coroutines for Concurrency	36
Chapter 8: Embedding and Synthesis	44
Embedding Lua in a Host Environment	44
Review of Key Concepts	49

Chapter 5: Metatables and OOP

Tables as Sets

Lecture

Why this comes first

Chapter 4 ended on the idea that the table is one structure wearing many hats. This lecture puts on the last hat before the chapter's real subject: the table as a **set**, where the question is not "what is stored at position 3" but "is this thing a member or not." It is deliberately a gentle lecture. It reuses exactly three Chapter 4 facts (keys of any type, `nil` as deletion, and `pairs` walking every key), and then, at the very end, it shows you the one thing a plain table cannot do for a set: let you write `a + b` to mean "union." That gap is what the "Metatables Basics" lecture's metatables close.

By the end you should be able to:

- Build a set by storing members as keys with the value `true`, and test membership with a single index rather than a loop.
- Add, remove, and count members, and explain why `#` is the wrong tool for counting a set.
- Write union, intersection, and difference as ordinary functions.
- Build a bag (a multiset) by storing counts instead of `true`.
- Name the three traps that make a set silently wrong: `false` as a value, identity versus equality for table members, and float keys.

1. Members are keys, not values

The instinctive way to store "a collection of words" is a list, and the instinctive membership test is then a loop that compares every element. *Programming in Lua* names the better approach directly: "an efficient and simple way to represent such sets is to put the set elements as *indices* in a table. Then, instead of searching the table for a given element, you just index the table and test whether the result is `nil` or not" (PiL 1st ed., §11.5).

The book's own constructor turns a list into a set in three lines:

```
LUA
local function Set(list)
  local set = {}
  for _, l in ipairs(list) do set[l] = true end
  return set
end

local reserved = Set{"while", "end", "function", "local"}

print(reserved["while"]) --> true
print(reserved["banana"]) --> nil
```

Two Chapter 3 and Chapter 4 details are doing quiet work here. `Set{...}` is a call with a single table-constructor argument, so the parentheses can be dropped. And the membership test needs no loop at all, because an absent key reads as `nil`, which is falsy:

```
LUA
local word = "local"
if reserved[word] then
  print(word .. " is a reserved word")
end
```

The practical rule: if the question your code keeps asking is "is X in here?", store X as a key. A list answers that question by scanning every element; a set answers it with one index.

2. Adding, removing, and counting

Adding a member is an assignment of `true`. Removing one is an assignment of `nil`, which is Chapter 4's nil-as-deletion rule doing exactly the job it was built for: the manual defines a key associated with `nil` as not part of the table at all.

```
LUA
local tags = Set{"Lua", "tables"}

tags["metatables"] = true  -- add
tags["tables"] = nil       -- remove

print(tags["tables"])      --> nil, genuinely gone, not "present but empty"
```

Adding a member that is already present is harmless: it overwrites `true` with `true`. That is one of the properties that makes this a set rather than a list. Duplicates cannot exist, because a table has at most one entry per key.

Counting is where Chapter 4 pays off again. **# does not count a set.** The length operator is defined in terms of *borders* over positive integer keys, and a set of strings has no integer keys at all, so `#tags` is `0` no matter how many members it holds. The only reliable count is a walk over every key with `pairs`:

```
LUA
local function size(set)
  local n = 0
  for _ in pairs(set) do n = n + 1 end
  return n
end

print(#tags)      --> 0, the length operator sees no sequence here
print(size(tags))  --> 2
```

If a program asks for the size constantly, keep a count field beside the set and update it on every add and remove, the same bookkeeping Chapter 4 recommended for holey arrays. For most code, the walk is fine.

3. Union, intersection, difference

Set algebra is a handful of short loops. The versions below follow the shape of *Programming in Lua's* own `Set.union` and `Set.intersection` (PiL 1st ed., §13.1), written as local functions so they create no globals:

```
LUA
local function union(a, b)
  local res = {}
  for k in pairs(a) do res[k] = true end
  for k in pairs(b) do res[k] = true end
  return res
end

local function intersection(a, b)
  local res = {}
  for k in pairs(a) do
    if b[k] then res[k] = true end
  end
  return res
end

local function difference(a, b)
  local res = {}
  for k in pairs(a) do
    if not b[k] then res[k] = true end
  end
  return res
end
```

FA26 | 2026

```

return res
end

```

The book's own `intersection` is one line shorter: it writes `res[k] = b[k]` unconditionally, relying on the fact that assigning `nil` simply does not create the key. That trick is correct for a set whose values are all `true`, but it copies whatever value `b` happens to hold, so it quietly does the wrong thing on a bag (Section 4). The explicit `if` above costs one line and works for both.

Every function builds and returns a **new** table and leaves both inputs untouched. That matters because of Chapter 4's reference semantics: a function that added members to `a` directly would change the caller's set, since the caller and the function share the same table.

Printing a set also needs a small helper, because `pairs` order is arbitrary (Chapter 4, "Tables as Arrays and Dictionaries"). Sorting the keys first gives stable, testable output:

```

LUA
local function toString(set)
    local keys = {}
    for k in pairs(set) do keys[#keys + 1] = tostring(k) end
    table.sort(keys)
    return "{" .. table.concat(keys, ", ") .. "}"
end

local a = Set{"x", "y", "z"}
local b = Set{"y", "z", "w"}
print(toString(union(a, b)))    --> {w, x, y, z}
print(toString(intersection(a, b))) --> {y, z}
print(toString(difference(a, b))) --> {x}

```

4. Bags: counting instead of true

A **bag** (or multiset) is a set that remembers how many times each member was added. The change is small: store a count as the value instead of `true`. The first-edition page of *Programming in Lua* that introduces sets is titled "Sets and Bags," but its body covers only sets (checked directly for this lecture), so the bag below is built from Chapter 4's rules rather than quoted from the book.

```

LUA
local function bagInsert(bag, item)
    bag[item] = (bag[item] or 0) + 1
end

local function bagRemove(bag, item)
    local count = bag[item]
    if count == nil then return end    -- removing something absent does nothing
    if count == 1 then
        bag[item] = nil              -- last copy gone: delete the key entirely
    else
        bag[item] = count - 1
    end
end

local votes = {}
bagInsert(votes, "tea")
bagInsert(votes, "tea")
bagInsert(votes, "coffee")
bagRemove(votes, "coffee")

print(votes["tea"])    --> 2
print(votes["coffee"]) --> nil, not 0

```

Two idioms from earlier chapters carry the whole thing. `bag[item] or 0` is Chapter 2's default-value idiom, turning "absent" into "zero" for the addition. And removing the key when the count reaches zero, instead of storing `0`,

keeps the bag honest: a member with count `0` would still be visited by `pairs` and would still be truthy, since `0` is true in Lua (Chapter 1).

5. Three traps that make a set silently wrong

Trap 1: storing `false`. A set test is `if set[x] then`. If some code writes `set[x] = false` to "remove" a member, the key is still present: `pairs` still visits it and a size count still includes it, yet the membership test now says "absent." Remove with `nil`, never with `false`.

Trap 2: table members are compared by identity. Table keys follow the manual's *raw equality*: two keys name the same entry only if they are raw equal. Strings and numbers compare by value, so `"lua"` is always the same member. Tables compare by identity, so two separately constructed tables are two different members even when their contents match:

```
LUA
local seen = {}
seen[{1, 2}] = true
print(seen[{1, 2}]) --> nil, a DIFFERENT table with the same contents
```

If you need "same contents means same member," convert each value to a string key first (for example `"1,2"`) and store that instead.

Trap 3: float keys become integers. The Lua 5.4 manual states that "any float used as a key that is equal to an integer is converted to that integer," so `set[2.0]` and `set[2]` are one member, not two. That is usually what you want, but it means a set cannot tell the float `2.0` apart from the integer `2`. The manual also forbids `nil` and NaN as keys, so neither can ever be a set member.

6. The gap this lecture leaves open

Everything above works, and none of it needed anything beyond Chapter 4. But look at how the calls read:

```
LUA
local both = intersection(union(a, b), c)
```

For a mathematical object like a set, you might reasonably want to write `(a + b) * c` instead, and have `print(a)` show the members rather than an address like `table: 0x55d0c8a4`. A plain table cannot do that: `+` on two tables is an error, and `tostring` of a table only prints its address. The mechanism that lets a table say "when someone adds me to something, call this function" is the metatable, and Programming in Lua introduces metatables with exactly this `Set` example for exactly this reason. That is "Metatables Basics".

7. What to take away

A set is a table whose members are its keys, each mapped to `true`, and membership is a single index that returns `true` or `nil`. Add with `true`, remove with `nil`, and count with a `pairs` walk, never with `#`, which sees no sequence in a table keyed by strings. Union, intersection, and difference are short loops that build new tables and leave their inputs alone. A bag swaps `true` for a count and deletes the key when the count reaches zero. And the three silent traps are storing `false` instead of `nil`, expecting two equal-looking tables to be the same member, and forgetting that a float key with an integral value becomes an integer.

Next: metatables, the mechanism that lets these sets use `+`, `*`, `<=` and `print` directly.

Key Terms

- **Set as table:** a table whose keys are the members and whose values are all true, so membership is a single index that returns true or nil.

- **Membership test:** indexing the set with a candidate and testing the result, relying on the fact that an absent key reads as nil and nil is falsy.
- **Set size:** the number of members, which must be counted with a pairs walk because the length operator only measures integer-keyed sequences and reports 0 for a set of strings.
- **Set union:** a new set containing every key of either input, built by two pairs loops without modifying either input.
- **Set intersection:** a new set containing only keys present in both inputs, built by one pairs loop with a membership test on the second set.
- **Set difference:** a new set containing the keys of the first input that are absent from the second.
- **Bag:** a multiset that stores a count per member instead of true, incrementing with $\text{bag}[x] = (\text{bag}[x] \text{ or } 0) + 1$ and deleting the key when the count reaches zero.
- **False is not removal:** assigning false keeps the key in the table so pairs still visits it, while the membership test reports it absent; remove members with nil.
- **Identity keys:** tables used as set members compare by identity, so two separately built tables with equal contents are two different members.
- **Float key normalisation:** Lua 5.4 converts any float key equal to an integer into that integer, so 2.0 and 2 are the same set member.

Summary

A set is a table whose members are its keys, each mapped to true, so membership is one index returning true or nil. Add with true, remove with nil, and count with a pairs walk, never with #. Union, intersection and difference are short loops that return new tables and leave their inputs alone. A bag stores counts and deletes a key when its count reaches zero. The silent traps are storing false instead of nil, expecting equal looking tables to be one member, and forgetting that an integral float key becomes an integer. A plain table still cannot make + or print work on a set; metatables can.

Metatables Basics

Lecture

Why this comes next

"Tables as Sets" ended with a set that could not use +, and a table that printed as an address. This lecture is the conceptual cliff the whole book has been walking toward, and it is smaller than its reputation. A metatable is an ordinary table. Lua consults it only at the moments when it would otherwise not know what to do: adding two tables, reading a key that is not there, calling something that is not a function. The authors' history paper describes the fallbacks that metatables later replaced in exactly those terms: "a more general fallback mechanism to let the programmer intervene whenever Lua did not know how to proceed" (Ierusalimsky et al., HOPL III). Hold onto that sentence and every rule below follows from it.

By the end you should be able to:

- Attach and inspect a metatable with `setmetatable` and `getmetatable`.
- Explain when Lua looks for a metamethod and which operand's metatable it checks first.
- Use `__index` as a table and as a function, for defaults and delegation.
- Use `__newindex` to intercept writes, including a read-only table.
- Use `rawget`, `rawset`, `rawequal`, and `rawlen` to step around metamethods, and explain why a metamethod usually needs them.
- Give the "Tables as Sets" set working +, *, and print.

1. A metatable is an ordinary table

The Lua 5.4 manual opens section 2.4 with the definition:

"Every value in Lua can have a *metatable*. This *metatable* is an ordinary Lua table that defines the behavior of the original value under certain events."

The keys of that table are strings naming events, each "prefixed by two underscores," and the value stored under one is called a *metavalue*. When the metavalue is a function, it is called a *metamethod*. So `__add` is the key, and the function stored under it is the `__add` metamethod.

New tables start with none. *Programming in Lua* shows this directly (PiL 1st ed., ch. 13):

```
LUA
local t = {}
print(getmetatable(t))  --> nil
```

`setmetatable(t, mt)` attaches one. Confirmed from the manual text and the 5.4 source (`lbaselib.c`), it accepts only a table as its first argument, a table or `nil` as its second (`nil` removes the metatable), and **returns the first argument**. That return value is why creation and attachment fit in one line:

```
LUA
local mt = {}
local obj = setmetatable({}, mt)  -- create, attach, and keep, in one step
print(getmetatable(obj) == mt)   --> true
```

Three facts about who owns a metatable, all from the manual:

- "Tables and full userdata have individual metatables, although multiple tables and userdata can share their metatables." Sharing is the normal case: one metatable describes a whole family of tables.
- "Values of all other types share one single metatable per type." All strings share one, which is why `("abc"):upper()` works: the string library "sets a metatable for strings where the `__index` field points to the `string` table."
- From Lua code you can only set the metatable of a table. Changing the metatable of any other type needs the debug library or the C API.

2. When Lua looks, and where

A metamethod is never called on the normal path. It is called only when the primitive operation cannot proceed.

The manual spells this out event by event:

- `__add`: "If any operand for an addition is not a number, Lua will try to call a metamethod." The same rule covers `__sub`, `__mul`, `__div`, `__mod`, `__pow`, `__unm`, and `__idiv`.
- `__concat`: tried "if any operand is neither a string nor a number."
- `__lt` and `__le`: tried "only when the values being compared are neither both numbers nor both strings."
- `__eq`: tried "only when the values being compared are either both tables or both full userdata and they are not primitively equal." Comparing a table with itself never calls `__eq`.
- `__index` and `__newindex`: only when the key "is not present in `table`." A key that exists is read and written directly.

Which metatable? For a binary operator, Lua checks the first operand's metatable, then the second's, and raises an error if neither has the field. The 5.4 source (`ltm.c`) does exactly that, with comments that read "try first operand" and "try second operand." *Programming in Lua* states the same rule for `__add`, adding that the first operand's metamethod is used "independently of the second value" (PiL 1st ed., §13.1). So `set + 1` and `1 + set` both reach the set's `__add`, and your metamethod has to be ready for either argument to be the non-set.

One more manual sentence explains a later surprise: "Lua queries metamethods in metatables using a raw access." A metamethod must be a real field of the metatable itself. "Object-Oriented Programming with Metatables" shows why that matters for inheritance.

3. Caching in the "Tables as Sets" set

Here is the "Tables as Sets" lecture's set, rebuilt so every set shares one metatable. This is the same design *Programming in Lua* uses to introduce metatables (§13.1).

```
LUA
local Set = {}
local mt = {}          -- one metatable shared by every set

function Set.new(list)
    local set = setmetatable({}, mt)
    for _, v in ipairs(list) do set[v] = true end
    return set
end

function Set.union(a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

function Set.intersection(a, b)
    local res = Set.new{}
    for k in pairs(a) do
        if b[k] then res[k] = true end
    end
    return res
end

function Set.tostring(set)
    local keys = {}
    for k in pairs(set) do keys[#keys + 1] = tostring(k) end
    table.sort(keys)
    return "{" .. table.concat(keys, ", ") .. "}"
end

mt.__add = Set.union
mt.__mul = Set.intersection
mt.__tostring = Set.tostring

local a = Set.new{10, 20, 30}
local b = Set.new{30, 1}
print(a + b)      --> {1, 10, 20, 30}
print((a + b) * b) --> {1, 30}
```

`print` now shows members because of `__tostring`. The manual's entry for `tostring` says that if the metatable "has a `__tostring` field, then `tostring` calls the corresponding value with `v` as argument, and uses the result of the call as its result." The 5.4 source (`luaolib.c`) adds a rule the manual's entry does not state: the result must be a string or a number, or Lua raises '`__tostring` must return a string'. A number passes the check, and `tostring` hands it back as a string. In 5.4, `print` no longer calls the global `tostring` function (manual §8.2) but still honours `__tostring`, so overriding the global `tostring` will not change what `print` shows, while setting `__tostring` will.

The manual also recommends an order: "add all needed metamethods to a table before setting it as a metatable of some object." The example follows it where it counts: all three fields are filled in before the first `Set.new` call runs, so no set ever exists with a half-built metatable.

4. `__index`: what to do about a missing key

`__index` is the metamethod you will use most, and it comes in two forms. The manual: "The metavalue for this event can be either a function, a table, or any value with an `__index` metavalue."

The **function form** is called with the table and the key, and its result is the value of the lookup. *Programming in Lua* uses it for default values (PiL 1st ed., §13.4.3):

```
LUA
local function setDefault(t, d)
    setmetatable(t, { __index = function() return d end })
end

local scores = { anne = 90 }
setDefault(scores, 0)
print(scores.anne) --> 90, present, so __index is never consulted
print(scores.bob)  --> 0, absent, so __index answers
```

The book then points out that this builds one metatable per table, which "may be expensive if we have many tables that need default values," and offers a shared version: store each table's default under a private key and let one metatable read it back.

```
LUA
local key = {} -- a table used as a key is unique, so nothing can collide with it
local defaultsMt = { __index = function(t) return t[key] end }

local function setDefault(t, d)
    t[key] = d
    setmetatable(t, defaultsMt)
end
```

The **table form** is simpler and cheaper. When the metavalue is a table, "the final result is the result of indexing this metavalue with `key`." In other words, a missing key is looked up in a second table instead:

```
LUA
local prototype = { x = 0, y = 0, width = 100, height = 100 }
local windowMt = { __index = prototype }

local w = setmetatable({ x = 10, y = 20 }, windowMt)
print(w.x, w.width) --> 10 100, x is w's own; width comes from prototype
```

This is *Programming in Lua*'s `Window` example (§13.4.1), and the book calls the table form "a cheap and simple way of implementing single inheritance." The manual adds that this second lookup "is regular, not raw, and therefore can trigger another `__index` metavalue." That is how chains form: if `prototype` itself had a metatable with `__index`, a miss in `prototype` would continue into the next table. "Object-Oriented Programming with Metatables" builds classes and subclasses out of exactly this.

5. `__newindex`: what to do about a new key

`__newindex` is the write-side twin. It fires on `t[key] = value` under the same condition as `__index`: the key is not already present. The manual:

"Whenever a `__newindex` metavalue is invoked, Lua does not perform the primitive assignment. If needed, the metamethod itself can call `rawset` to do the assignment."

That sentence is the whole design space. A `__newindex` function can refuse the write, redirect it, log it, or validate it, and nothing is stored unless it chooses to store it. *Programming in Lua's* read-only table uses refusal (§13.4.5):

```
LUA
local function readOnly(t)
    local proxy = {}
    local mt = {
        __index = t,
        __newindex = function(_, _, _)
            error("attempt to update a read-only table", 2)
        end,
    }
    return setmetatable(proxy, mt)
end

local days = readOnly{"Sunday", "Monday", "Tuesday"}
print(days[1])      --> Sunday
days[2] = "Noday"   --> error: attempt to update a read-only table
```

Notice the proxy is **empty**, and it has to be. Both metamethods fire only for absent keys, so if the data lived in `proxy` itself, reads and writes to existing keys would skip the metamethods entirely. Keeping the real data in `t` and handing out an empty `proxy` guarantees that every access passes through. The `2` passed to `error` points the error message at the caller's line rather than at the metamethod.

This protection has limits worth stating plainly. `rawset(days, 2, "x")` writes straight into the proxy, and `#days` and `pairs(days)` see the empty proxy rather than `t`. It is a guard against mistakes, not a security boundary.

6. The raw functions, and the recursion they prevent

Four functions do the primitive operation with metamethods switched off. Confirmed against the 5.4 manual text and source:

- `rawget(t, k)` gets "the real value of `table[index]`, without using the `__index` metavalue."
- `rawset(t, k, v)` sets it "without using the `__newindex` metavalue," and returns `t`.
- `rawequal(a, b)` checks equality "without invoking the `__eq` metamethod."
- `rawlen(v)` returns the length of a table or string "without invoking the `__len` metamethod."

The everyday reason to reach for them is recursion. A `__newindex` that stores with a normal assignment re-enters itself, because the key is still absent at that moment:

```
LUA
local logged = setmetatable({}, {
    __newindex = function(t, k, v)
        print("setting " .. tostring(k))
        rawset(t, k, v)      -- t[k] = v here would call __newindex again, forever
    end,
})

logged.name = "Anne"      --> setting name
logged.name = "Kim"       -- prints nothing: the key now exists, so no metamethod
print(rawget(logged, "name")) --> Kim
```

The second assignment printing nothing is the Section 2 rule again: once `rawset` created the key, `__newindex` stopped being consulted for it. If you need to see every write, use the empty-proxy pattern from Section 5.

7. Protecting a metatable with `__metatable`

A metatable can hide and lock itself. Confirmed from the manual and `lbaselib.c`: if the metatable has a `__metatable` field, `getmetatable` returns that field's value instead of the metatable, and `setmetatable` raises an error rather than replace it.

```
LUA
local mt = { __metatable = "Locked" }
local t = setmetatable({}, mt)
print(getmetatable(t))    --> locked
setmetatable(t, {})      --> error: cannot change a protected metatable
```

Programming in Lua shows the same behaviour with a slightly older error text ("cannot change protected metatable"); the 5.4 source reads "cannot change a protected metatable." Match on the idea, not the exact wording, if you ever test for it.

8. What to take away

A metatable is an ordinary table of event handlers, attached with `setmetatable` (which returns the table) and read with `getmetatable`. Lua consults it only when the primitive operation cannot proceed: an operand that is not a number, a key that is not present, a value that is not a function. For binary operators it tries the first operand's metatable, then the second's. `__index` answers reads of missing keys, either by calling a function or by looking in another table, and that second lookup can itself chain. `__newindex` intercepts writes of new keys and stores nothing unless it calls `rawset`. The `raw` functions bypass metamethods and are how a metamethod avoids calling itself. And `__metatable` hides and locks a metatable.

Next: the table form of `__index`, the colon sugar from Chapter 3, and one shared metatable per family add up to classes, constructors, and inheritance, with no `class` keyword anywhere.

Key Terms

- **Metatable:** an ordinary table attached to a value that defines how the value behaves under events such as addition or indexing a missing key.
- **Metamethod:** a function stored in a metatable under an event key made of two underscores and the event name such as `__add` or `__index`.
- **setmetatable:** attaches a metatable to a table or removes it with `nil` and returns the table so creation and attachment fit in one expression.
- **getmetatable:** returns a value's metatable or `nil`, or the `__metatable` field's value when the metatable is protected.
- **Metamethod dispatch:** Lua consults a metatable only when the primitive operation cannot proceed, and for binary operators checks the first operand's metatable before the second's.
- **__index:** consulted when a key is absent; as a function it is called with the table and key, as a table the lookup repeats in that table and can chain further.
- **__newindex:** consulted when assigning to an absent key; Lua performs no primitive assignment, so the metamethod must call `rawset` if the value should be stored.
- **rawget and rawset:** read and write a table field while bypassing `__index` and `__newindex`, used inside metamethods to avoid infinite recursion.
- **rawequal and rawlen:** compare without `__eq` and measure a table or string without `__len`.
- **__tostring:** the metamethod `tostring` and `print` use to format a value, which must return a string or a number, or Lua raises an error.
- **__metatable:** a metatable field that makes `getmetatable` return its value instead and makes `setmetatable` raise an error.
- **Proxy table:** an empty table whose metatable forwards reads and writes elsewhere, empty so that every access is an absent-key access that reaches the metamethods.

Summary

A metatable is an ordinary table of event handlers attached with `setmetatable`, which returns the table, and read with `getmetatable`. Lua consults it only when a primitive operation cannot proceed, and for binary operators it tries the first operand's metatable and then the second's. `__index` answers reads of absent keys by calling a function or looking in another table, a lookup that can chain. `__newindex` intercepts writes of absent keys and stores nothing unless it calls `rawset`. The raw functions bypass metamethods, `__tostring` controls printing, and `__metatable` locks a metatable.

Object-Oriented Programming with Metatables

Lecture

Why this comes last

Chapter 3 previewed the colon sugar and promised it would matter later. "Metatables Basics" gave you `__index` as a table, a second place to look when a key is missing. This lecture adds nothing new to the language. It combines those two pieces, plus one shared metatable per family of tables, and the result is a complete object system: classes, constructors, methods, inheritance, overriding, and operator overloading. The point to carry out of it is that Lua's objects are not a feature bolted onto the language. They are a *consequence* of metatables, and once you can see the lookup chain, nothing about them is magic.

By the end you should be able to:

- Explain what `self` is and what the colon does in both a definition and a call.
- Write a class with the `Class.__index = Class` idiom and a constructor.
- Trace a method call through the `__index` chain from instance to class.
- Build a subclass that inherits, overrides, and calls its parent's method.
- Overload operators on a class, and explain why metamethods are not inherited through `__index`.
- Say why Lua ships no `class` keyword.

1. The colon, cashed in

The manual defines both halves of the sugar (§3.4.10 and §3.4.11):

"A call `v:name(args)` is syntactic sugar for `v.name(v,args)`, except that `v` is evaluated only once."

"Thus, the statement `function t.a.b.c:f (params) body end` is syntactic sugar for `t.a.b.c.f = function (self, params) body end`."

Programming in Lua summarises the pair in one sentence: "The effect of the colon is to add an extra hidden parameter in a method definition and to add an extra argument in a method call" (PiL 1st ed., ch. 16). The hidden parameter is named `self`.

Why bother? The book's motivating example is a method that names its own table:

```
LUA
Account = { balance = 0 }
function Account.withdraw(v)
    Account.balance = Account.balance - v    -- hard-wired to the global name
end
```

That works exactly until the table is renamed or copied: after `a = Account; Account = nil`, calling `a.withdraw(100)` fails because the body still reaches for `Account`. The fix is to operate on whatever table the method was called on, which is what `self` is:

```
LUA
local Account = { balance = 0 }

function Account:withdraw(v)    -- same as Account.withdraw = function(self, v)
    self.balance = self.balance - v
end

Account:withdraw(100)           -- same as Account.withdraw(Account, 100)
print(Account.balance)         --> -100
```

The rule to memorise: define with `:` and call with `:`, or define with `.` and pass the object yourself. Mixing them (defining with `:` and calling with `.`) shifts every argument by one, and the error can surface somewhere far from the call.

2. A class is a metatable that points at itself

One table can hold methods, but we want many accounts that share those methods and each hold their own balance. "Metatables Basics" already has the tool: give every account the same metatable, and make that metatable's `__index` the table of methods. The idiom that makes this compact is to let one table play both roles:

```
LUA
local Account = {}
Account.__index = Account    -- the class is its own instances' __index

function Account.new(balance)
    local self = setmetatable({}, Account)
    self.balance = balance or 0
    return self
end

function Account:deposit(v)
    self.balance = self.balance + v
end

function Account:getBalance()
    return self.balance
end

local a = Account.new(50)
a:deposit(25)
print(a:getBalance())       --> 75
```

Trace `a:deposit(25)` step by step, because this is the whole mechanism:

1. The colon turns it into `a.deposit(a, 25)`.
2. `a` has no `deposit` key of its own, so the read of `a.deposit` misses.
3. `a`'s metatable is `Account`, and `Account.__index` is `Account`, so Lua looks up `deposit` in `Account` and finds the function.
4. The function runs with `self` bound to `a`, so it updates `a.balance`.

Programming in Lua describes the same lookup as `getmetatable(a).__index.deposit(a, 100.00)` (PiL 1st ed., §16.1). The instance holds data; the class holds behaviour; `__index` connects them.

Reads fall through, writes do not. Suppose the class holds a default: `Account.balance = 0`, and an instance never set its own. Reading `b.balance` falls through to `Account` and returns `0`. But `b.balance = b.balance + v` writes to `b`,

and because `Account` defines no `__newindex`, that write creates a new field in `b` itself. From then on `b` has its own balance. The book makes this exact point about inherited field defaults.

The trap that follows from it: a mutable default shared through the class is shared by every instance. If `Account.history = {}` and a method does `table.insert(self.history, v)`, every account appends to the *same* table, because the read of `self.history` falls through and nothing is ever written to the instance. Create per-instance tables in the constructor.

3. The Programming in Lua constructor, and why it looks odd

The book's constructor is written as a method on the class, and the book's class starts with a class-level default balance, which the inheritance example below relies on:

```
LUA
Account.balance = 0          -- class-level default, as in the book's Account = {balance = 0}

function Account:new(o)
  o = o or {}                -- create a table if the caller did not pass one
  setmetatable(o, self)
  self.__index = self
  return o
end

local b = Account:new{ balance = 10 }
```

Two lines deserve a comment. `o = o or {}` is Chapter 2's default idiom, and it lets the caller pass initial fields as a table (Chapter 3's named-arguments pattern). `self.__index = self` runs on *every* construction, which looks redundant for `Account` but is what makes the next section work: when a subclass calls this inherited `new`, `self` is the subclass, so the subclass gets its own `__index` set without writing anything extra.

Both styles work. The `Account.new` style from Section 2 is explicit and easy to read; the `Account:new` style from the book is shorter to extend. Pick one per codebase and stay with it.

4. Inheritance is a longer `__index` chain

A subclass is a class whose own missing keys fall through to its parent. In the book's style, the subclass is literally created as an instance of the parent (PiL 1st ed., §16.2):

```
LUA
local SpecialAccount = Account:new()  -- an "instance" of Account

function SpecialAccount:withdraw(v)  -- override
  if v - self.balance >= self:getLimit() then
    error("insufficient funds")
  end
  self.balance = self.balance - v
end

function SpecialAccount:getLimit()
  return self.limit or 0
end

local s = SpecialAccount:new{ limit = 1000 }
s:deposit(100)      -- found in Account
s:withdraw(200)     -- found in SpecialAccount, the override
print(s.balance)    --> -100
```

`SpecialAccount:new{...}` finds `new` in `Account`, but runs it with `self` set to `SpecialAccount`, so `s` gets `SpecialAccount` as its metatable. The chain for `s:deposit` is: `s` (miss), then `SpecialAccount` (miss), then `Account` (hit). The chain for `s:withdraw` stops one step earlier, at the override. That is all overriding is: the nearest table in the chain wins.

Calling the parent's version from an override uses a plain dot call with `self` passed explicitly, since the colon would find the override again:

```
LUA
function SpecialAccount:deposit(v)
    print("special deposit " .. v)
    Account.deposit(self, v)    -- the parent's method, on this object
end
```

The explicit-style equivalent of `Account:new()` for building a subclass is two lines:

```
LUA
local Savings = setmetatable({}, { __index = Account }) -- misses go to Account
Savings.__index = Savings                             -- instances look here first
```

Either way, the book also notes that a single object can override a method just for itself, by assigning a function to that object's own field. The chain simply finds it one step sooner.

5. Operators on a class, and the inheritance trap

Because the class table *is* the metatable of its instances, operator metamethods go straight into the class:

```
LUA
local Point = {}
Point.__index = Point

function Point.new(x, y)
    return setmetatable({ x = x, y = y }, Point)
end

function Point.__add(a, b) return Point.new(a.x + b.x, a.y + b.y) end
function Point.__eq(a, b) return a.x == b.x and a.y == b.y end
function Point.__tostring(p) return "(" .. p.x .. ", " .. p.y .. ")" end

local p = Point.new(1, 2) + Point.new(3, 4)
print(p)                --> (4, 6)
print(p == Point.new(4, 6)) --> true, via __eq
print(rawequal(p, Point.new(4, 6))) --> false, two different tables
```

`__call` makes the class itself callable, which is a common way to get `Point(1, 2)` as constructor syntax. It must go on the class's *own* metatable, since `__call` is looked up on the value being called:

```
LUA
setmetatable(Point, {
    __call = function(_, x, y) return Point.new(x, y) end,
})
print(Point(7, 8))    --> (7, 8)
```

The trap: metamethods are **not** inherited through `__index`. The manual says "Lua queries metamethods in metatables using a raw access," so Lua asks `rawget(mt, "__tostring")`, not `mt.__tostring`. A subclass's instances find ordinary methods through the chain, but their `__tostring` lookup checks only the subclass table itself and misses:

```
LUA
local Point3 = setmetatable({}, { __index = Point })
Point3.__index = Point3

function Point3.new(x, y, z)
    local p = setmetatable(Point.new(x, y), Point3)
    p.z = z
    return p
end

local q = Point3.new(1, 2, 3)
```

```
print(q.x)      --> 1, ordinary fields and methods still reach Point
print(q)        --> table: 0x..., Point.__tostring was NOT found
```

The fix is to copy the metamethods you want into each subclass:

```
LUA
Point3.__tostring = Point.__tostring
Point3.__eq = Point.__eq
Point3.__add = Point.__add
```

6. Define `__le` yourself

Comparison operators need care in 5.4. The manual's list of incompatibilities with Lua 5.3 (§8.1) states: "The use of the `__lt` metamethod to emulate `__le` has been removed. When needed, this metamethod must be explicitly defined."

Programming in Lua describes the older world, where `a <= b` could be answered as `not (b < a)`, and explains why that is wrong for partial orders such as set containment, where both can be false (§13.2).

There is a wrinkle worth knowing so it does not mislead you. The 5.4 source keeps the old fallback behind a compile-time switch, `LUA_COMPAT_LT_LE`, and the stock `src/Makefile` in the official 5.4.9 distribution builds with `-DLUA_COMPAT_5_3`, which turns it on. So on a build made from that Makefile, code that defines only `__lt` will appear to support `<=`, and then fail on a build without the compatibility flag. **Always define `__lt` and `__le` explicitly.** For `>` and `>=` nothing extra is needed: Lua translates `a > b` to `b < a` and `a >= b` to `b <= a`.

7. Why Lua has no class keyword

Programming in Lua states it flatly: "Lua does not have the concept of class; each object defines its own behavior and has a shape of its own" (§16.1). The authors' history paper explains that this was a deliberate choice, made early and kept despite pressure (Ierusalimsky et al., HOPL III, §5.2):

"we did not want to turn Lua into an object-oriented language because we did not want to fix a programming paradigm for Lua ... Instead of fixing a model, we decided to provide flexible mechanisms that would allow the programmer to build whatever model was suitable to the application."

That is the Chapter 1 design brief showing up again. lua.org's About page states it this way: "A fundamental concept in the design of Lua is to provide meta-mechanisms for implementing features, instead of providing a host of features directly in the language." The table plus the metatable turns out to be enough mechanism for classes, prototypes, or read-only records. The practical cost is that a Lua codebase has to pick its own conventions. When you read someone else's Lua, find their constructor and their `__index` line first; they show how objects are built and where methods are found.

8. What to take away

The colon adds a hidden `self` parameter in a definition and passes the object as the first argument in a call. A class is a table of methods that is also its instances' metatable, joined by `Class.__index = Class`. A method call on an instance misses on the instance, falls through `__index` to the class, and runs with `self` bound to the instance. Reads fall through to the class, writes land on the instance, and a mutable default stored on the class is shared by every instance. Inheritance is a longer `__index` chain, overriding is "nearest table wins," and a parent's method is called with a dot and an explicit `self`. Operator metamethods live in the class table but are looked up raw, so subclasses must copy them. Define `__le` yourself, because 5.4 removed the `__lt` fallback even where the stock build still hides that removal.

That is Chapter 5. Chapter 6 moves to the standard library and modules, where the same tables turn up once more as the thing a `require` returns.

Key Terms

- **self**: the hidden first parameter a colon definition adds, bound to whatever object the method was called on.
- **Colon sugar**: `v:name(args)` means `v.name(v, args)` with `v` evaluated once, and function `t:f(params)` means `t.f = function(self, params)`.
- **Class table**: a table of methods that also serves as the metatable of its instances, joined to them by `Class.__index = Class`.
- **Constructor**: a function that creates a table and attaches the class as its metatable, conventionally named `new` and often written `return setmetatable(obj, Class)`.
- **Method lookup**: a method call misses on the instance, falls through `__index` to the class, and runs with `self` bound to the instance.
- **Reads fall through and writes do not**: reading a missing field returns the class default, but assigning it creates a field on the instance because the class defines no `__newindex`.
- **Shared mutable default**: a table stored on the class is shared by every instance that never assigns its own, so per-instance tables belong in the constructor.
- **Inheritance chain**: a subclass whose own missing keys fall through to its parent via `__index`, so lookup walks instance then subclass then parent.
- **Method overriding**: defining a method on the subclass so the nearest table in the `__index` chain wins.
- **Calling the parent method**: `Parent.method(self, args)` with a dot and an explicit `self`, since a colon call would find the override again.
- **Metamethods are not inherited**: Lua looks metamethods up with raw access, so a subclass must copy `__tostring` and other operator metamethods from its parent.
- **__call constructor**: a `__call` metamethod on the class's own metatable that lets `Point(1, 2)` style calls create instances.
- **__le must be explicit**: Lua 5.4 removed emulating `__le` with `__lt`, although stock builds with `LUA_COMPAT_5_3` still do it, so define both `__lt` and `__le`.
- **No class keyword**: Lua deliberately provides mechanisms rather than a fixed object model, leaving each codebase to choose its own conventions.

Summary

The colon adds a hidden `self` parameter to a definition and passes the object as the first argument of a call. A class is a table of methods that is also its instances' metatable, joined by `Class.__index = Class`, so a method call misses on the instance and falls through to the class. Reads fall through; writes land on the instance; mutable defaults on the class are shared. Inheritance is a longer `__index` chain, overriding is nearest table wins, and a parent method is called with a dot and explicit `self`. Operator metamethods live in the class but are looked up raw, so subclasses copy them, and `__le` should always be defined explicitly.

Chapter 6: Standard Library and Modules

Introduction to the Standard Library

Lecture

Why this comes first

Lua's standard library is short. The 5.4 manual lists ten libraries (section 6): basic, coroutine, package, string, utf8, table, math, io, os and debug. It also says where the work goes: "The standard Lua libraries provide useful functions that are implemented in C through the C API." The interpreter with all of them built in takes 293K on 64-bit Linux (lua.org, About; see "Lua Basics").

So this lecture is a tour of a small box. The skill is knowing what is in it, so you stop reaching for something that is not.

By the end you should be able to:

- Explain why the standard libraries are ordinary tables.
- Slice, search and format strings with the `string` functions.
- Write Lua patterns and say how they differ from regular expressions.
- Use `gsub` with a string, a table and a function as the replacement.
- Use `table`, `math`, `os` and `io` for sorting, numbers, time and files.
- Recognise a few Lua 5.4 library details that are easy to miss.

1. The box is made of tables

The manual states: "Except for the basic and the package libraries, each library provides all its functions as fields of a global table or as methods of its objects." There is no import syntax: `math.sin` is syntactic sugar for `math["sin"]`, an ordinary table index (manual, §3.2). The standalone interpreter includes all standard libraries (section 7), so `string`, `table`, `math`, `os` and `io` are tables in the global environment, and Chapter 4's table skills apply as is.

Strings get one trick. The manual says the string library "sets a metatable for strings where the `__index` field points to the `string` table. Therefore, you can use the string functions in object-oriented style." That is Chapter 5's `__index`, cashed in:

```
LUA
print(getmetatable("").__index == string)  --> true
```

2. Strings: slicing, case, repetition

Indices start at 1 and may be negative: "the last character is at position -1, and so on." Strings are immutable ("immutable sequences of bytes"), so every function returns a new one.

```
LUA
local s = "Hello, Lua"
print(#s, s:sub(1, 5), s:sub(-3))  --> 10 Hello Lua
print(s:sub(8, 100))               --> Lua
print("[ " .. s:sub(5, 2) .. "]" ) --> [ ]
print(s:upper(), s:reverse())      --> HELLO, LUA auL ,olleH
print(s:byte(1, 3))               --> 72 101 108
print(string.char(72, 105))        --> Hi
```

Bad ranges are corrected, not rejected: "If j is greater than the string length, it is corrected to that length. If, after these corrections, i is greater than j, the function returns the empty string." Like Python slicing, that is forgiving and easy to get silently wrong.

3. Lua patterns are not regular expressions

Lua ships no regex engine. *Programming in Lua* explains: "Unlike several other scripting languages, Lua does not use POSIX regular expressions (regex) for pattern matching. The main reason for this is size: A typical implementation of POSIX regex takes more than 4,000 lines of code. This is bigger than all Lua standard libraries together. In comparison, the implementation of pattern matching in Lua has less than 500 lines." (PiL 1st ed., §20.1) Its key limit is in the manual's list of pattern items: a repetition applies to "a single character class followed by '*' (and likewise '+', '-' and '?'), never to a group (§6.4.1). So `(ab)*` does not work. Here is the result for alternation:

```
LUA
print(("cat"):match("cat/dog"))    --> nil
print(("cat/dog"):match("cat/dog")) --> cat|dog
```

The bar is a literal character. For "cat or dog", call `match` twice.

Classes (manual 6.4.1): `%a` "represents all letters," `%d` digits, `%l` lowercase, `%u` uppercase, `%s` space, `%w` alphanumeric, `%p` punctuation, `%x` hexadecimal, and a dot "represents all characters." The uppercase form is the complement: `%S` represents all non-space characters." Brackets make sets such as `[%w_]`, and `[^set]` is the complement.

Quantifiers apply to one class only. `*` matches zero or more, longest possible. `+` matches one or more. `-` "also matches sequences of zero or more characters in the class. Unlike '`*`', these repetition items will always match the shortest possible sequence." The trim idiom in section 5 uses that lazy `-?`. `?` matches zero or one.

Escaping uses a percent sign: `%x`: (where x is any non-alphanumeric character) represents the character x. This is the standard way to escape the magic characters." Never a backslash.

Anchors. A caret at the start of a pattern anchors the match at the beginning of the subject, and a dollar sign at the end anchors it at the end; elsewhere both "represent themselves." This lecture's text avoids typing the dollar sign, so the one example that needs it writes it as the decimal escape `"\36"`.

Captures use parentheses and are "numbered according to their left parentheses." The special capture `()` "captures the current string position (a number)." `%bxy` matches "strings that start with x, end with y, and where the x and y are balanced," so `%b()` finds balanced parentheses. `%f[set]`, the frontier, "matches an empty string at any position such that the next character belongs to set and the previous character does not."

4. The four pattern functions

`string.find` returns "the indices of s where this occurrence starts and ends," then any captures; a fourth argument `true` "turns off the pattern matching facilities." `string.match` returns the captures, or the whole match if there are none. `string.gmatch` "returns an iterator function," which is what Chapter 4's generic `for` wants. `string.gsub` returns "a copy of s in which all (or the first n, if given) occurrences of the pattern ... have been replaced," and "as its second value, the total number of matches."

```
LUA
local s = "Hello, Lua"
print(s:find("Lua"))           --> 8 10
print(s:find("(L)(u)"))       --> 8 9 L u
print(s:find(".", 1, true))    --> nil
print(s:match("(%a+), (%a+)")) --> Hello Lua
print(("flaaap"):find("(aa)")) --> 3 4 3 5
```

FA26 | 2026

```
print(("f(a(b)c d"):match("%b()")) --> (a(b)c)
print(("hello world"):gsub("o", "0")) --> hell0 w0rld 2
```

The plain search printed `nil`. The 5.4 manual calls the failure value `fail` and says "Currently, fail is equal to nil", so you test it the same way. A word counter with `gmatch`, sorted so the output is stable:

```
LUA
local counts = {}
for w in ("the cat and the hat and the bat"):gmatch("%a+") do
  counts[w] = (counts[w] or 0) + 1
end
local keys = {}
for k in pairs(counts) do keys[#keys + 1] = k end
table.sort(keys, function(a, b)
  if counts[a] ~= counts[b] then return counts[a] > counts[b] end
  return a < b
end)
for _, k in ipairs(keys) do io.write(k, "=", counts[k], " ") end
print() --> the=3 and=2 bat=1 cat=1 hat=1
```

5. gsub with a string, a table, a function

The manual defines three forms. A string: "%d", with d between 1 and 9, stands for the value of the d-th captured substring; the sequence %0 stands for the whole match." A table: "the table is queried for every match, using the first capture as the key." A function: "called every time a match occurs, with all captured substrings passed as arguments." For the last two, "if it is false or nil, then there is no replacement."

```
LUA
print(("hello world"):gsub("(%w+) (%w+)", "%2 %1")) --> world hello 1

local vars = { name = "Anne", lang = "Lua" }
print(("{name} writes {lang} and {missing}"):gsub("{(%w+)}", vars))
--> Anne writes Lua and {missing}

print(("3 + 4"):gsub("%d+", function(n) return tostring(tonumber(n) * 2) end))
--> 6 + 8
```

The extra parentheses drop the match count so `print` shows only the string. `{missing}` survives because the lookup returned `nil`.

Programming in Lua's trim idiom (PiL 1st ed., §20.3) shows why lazy `-` matters: `.-` expands as little as possible, so the two `%s*` swallow all the whitespace.

```
LUA
local END = "\36" -- the end anchor character, as a decimal escape
local function trim(s) return (s:match("^%s*(.-)%s*" .. END)) end
print("[ " .. trim(" hi there \t\n") .. "]") --> [hi there]
```

6. string.format

It follows C's `sprintf`, with differences the manual lists: "F, n, *, h, L, and l are not supported" and there is an extra `q`, which writes "a valid constant in Lua source code." Width and precision are "limited to two digits."

```
LUA
print(string.format("%d items, %s, %5.2f, %05d, %x", 3, "ok", math.pi, 42, 255))
--> 3 items, ok, 3.14, 00042, ff
print(string.format("%q", 'say "hi"')) --> "say \"hi\""
```

The 5.4 manual says `d` expects "an integer." A float with a fractional part has no integer representation, so `pcall(string.format, "%d", 1.5)` returns `false` with the message `bad argument #2 to 'string.format' (number has no integer representation)`, while `string.format("%d", 3.0)` gives `3`. Pass integers to `%d`.

7. table

`table.insert(t, x)` appends ("The default value for pos is #list+1"), `table.remove` removes the last element by default, `table.concat` joins with an optional separator, and `table.sort` sorts "in-place."

```
LUA
local t = {10, 20, 30}
table.insert(t, 40)
table.insert(t, 1, 5)
print(table.concat(t, " "))      --> 5, 10, 20, 30, 40
print(table.remove(t), table.remove(t, 1))  --> 40 5

local names = { "mia", "Zed", "anne", "Bob" }
table.sort(names)
print(table.concat(names, " "))  --> Bob Zed anne mia
table.sort(names, function(a, b) return a:lower() < b:lower() end)
print(table.concat(names, " "))  --> anne Bob mia Zed
```

The default order is `<`, so uppercase sorts first. The manual says the order function "receives two list elements and returns true when the first element must come before the second in the final order." The manual adds two rules that prevent real bugs: the function "must define a strict weak order," and "The sort algorithm is not stable: Different elements considered equal by the given order may have their relative positions changed." Break ties with a second key, as the word counter did. Mixed types fail: `pcall(table.sort, {3, "a", 1})` gave `false` attempt to compare string with number.

8. math

`math.floor` "returns the largest integral value less than or equal to x," `math.huge` is "a value greater than any other numeric value," and `math.fmod` "rounds the quotient towards zero" while `%` rounds the other way.

```
LUA
print(math.floor(3.3), math.floor(-3.3))      --> 3  -4
print(math.max(10.4, 7, -3, 20), math.min(3, 9, 4))  --> 20  3
print(math.fmod(-7, 3), -7 % 3)               --> -1  2
```

For random numbers, 5.4 says `math.random()` gives "a pseudo-random float with uniform distribution in the range [0,1)," and two integers give "a pseudo-random integer with uniform distribution in the range [m, n]." The 5.4 manual says the generator "now starts with a somewhat random seed" (section 8), so the `math.randomseed(os.time())` that PiL 1st ed. teaches (§18) is not needed, and "equal seeds produce equal sequences of numbers," which is how you get repeatable tests. One call is worth a closer look. The manual: "The call `math.random(0)` produces an integer with all bits (pseudo)random." Checking the subtype:

```
LUA
print(math.type(math.random(0)))  --> integer
```

9. os

`os.time{year=..., month=..., day=...}` builds a timestamp; "hour (default is 12)." Fields may be out of range: "if hour is 1000, it means 1000 hours after the time specified by the other fields," so `os.time` does date arithmetic for you.

```
LUA
local t0 = os.time{year = 2026, month = 1, day = 1, hour = 0}
local t1 = os.time{year = 2026, month = 1, day = 31, hour = 0}
print(os.difftime(t1, t0) / 86400)      --> 30.0
print(os.date("%*t", t0).wday)          --> 5
print(os.date("%Y-%m-%d", os.time{year = 2026, month = 1, day = 32, hour = 12}))
--> 2026-02-01
```

`os.date("!*t")` returns a table with `year`, `month`, `day`, `wday` ("Sunday is 1") and more; other formats follow "the ISO C function `strftime`." A leading `!` means Coordinated Universal Time, otherwise you get local time, so the same timestamp can print as different dates. 906000490 is the timestamp PiL 1st ed. uses in §22.1:

```
LUA
print(os.date("%d/%m/%Y", 906000490))    --> 16/09/1998  (local time, a zone behind UTC)
print(os.date("!*d/%m/%Y", 906000490))  --> 17/09/1998  (UTC)
```

Also: `os.clock` is "an approximation of the amount in seconds of CPU time used by the program," `os.getenv` returns the variable or a failure value, and `os.remove` and `os.rename` return a failure value plus message and code instead of raising.

10. io

The manual states one convention for every failing I/O function: "return fail on failure, plus an error message as a second result and a system-dependent error code as a third result." So `io.open` returns values rather than raising, and PiL 1st ed. (§21.2) teaches the guard `local f = assert(io.open(filename, mode))` and explains: "If the open fails, the error message goes as the second argument to `assert`, which then shows the message." Modes are `"r"` ("the default"), `"w"`, `"a"`, and `"r+"`, `"w+"`, `"a+"` for update.

```
LUA
print(io.open("/no/such/file.txt", "r"))
--> nil /no/such/file.txt: No such file or directory  2

local name = os.tmpname()
local f = assert(io.open(name, "w"))
f:write("line one\n", "line two\n", 3, "\n")
f:close()
for line in io.lines(name) do io.write("[", line, "]") end
print()    --> [line one][line two][3]
os.remove(name)
```

`io.lines(name)` is the idiomatic line reader; it "automatically closes the file" at the end but, if the file cannot be opened, "raises the error." `file:read` takes formats: `"n"` a number, `"l"` the next line ("the default format"), `"L"` the line with its newline, `"a"` the whole file ("this format never fails"), or a byte count. Close files yourself, since garbage collection closes them in "an unpredictable amount of time." `io.stdin`, `io.stdout` and `io.stderr` are never closed by the library.

11. Details worth knowing in 5.4

- `file:write`. In case of success it "returns file" (Lua 5.4 Reference Manual, `file:write`), so calls can be chained: `f:write("a"):write("b")`.
- `table.unpack`. The 5.4 manual defines `table.unpack (list [, i [, j]])` as returning "the elements from the given list." `table.pack` and `table.move` sit beside it in the table library. For example, `print(table.unpack({1, 2, 3}))` prints `1 2 3`.
- **Integers and `//`**. The 5.4 manual: "Floor division (`//`) is a division that rounds the quotient towards minus infinity," and float division "always" gives floats. So `7 // 2` is 3, `7 / 2` is 3.5, and `math.type(7 // 2)` is `integer`.
- `string.rep`. 5.4 documents an optional separator, `string.rep (s, n [, sep])`, so `("ab"):rep(3, "-")` gives `ab-ab-ab`.
- `utf8`. The `utf8` library works with UTF-8 encoded strings. `#` counts bytes, while `utf8.len` counts characters: for `s = "héllö"`, `#s` is 6 and `utf8.len(s)` is 5.

12. What to take away

The standard library is a few global tables, implemented in C. `string` slices with 1-based indices and searches with patterns that are not regexes: classes, single-class quantifiers, lazy `-`, captures, `%b`, `%f`. `gsub` takes a string, table or function and also returns a count. `table.sort` is unstable and needs a strict order. `os.time` normalises out-of-range fields. `io` functions return failure, message and code, so guard `io.open` with `assert`. Lua 5.4 has a few details worth remembering: `file:write` returns the file, unpacking is `table.unpack`, `//` is floor division, and `utf8.len` counts characters.

Next: splitting your own program into files, and what `require` does.

Key Terms

- **Library as table:** each standard library except `basic` and `package` is a global table whose functions are ordinary fields, so there is no import syntax.
- **String metatable:** all strings share a metatable whose `__index` points at the string table, which is why `s:upper()` and `string.upper(s)` are the same call.
- **Lua pattern:** Lua's own matching language, built from percent classes and single-class quantifiers, with no grouping under a modifier and no alternation operator.
- **Character class:** a percent letter such as `%a` for letters or `%d` for digits that matches one character, with the uppercase form meaning the complement.
- **Lazy quantifier:** the minus sign after a class, matching zero or more characters but as few as possible, which is what makes the trim idiom work.
- **Capture:** a parenthesised part of a pattern whose matched text is returned by `match` or `find` or passed to a `gsub` function, with `()` capturing a position.
- **Frontier pattern:** `%f[set]` matches an empty string where the next character is in the set and the previous one is not, useful for whole-word matches.
- **Balanced match:** `%bxy` matches text that starts with `x` and ends with the `y` that closes it, so `%b()` matches balanced parentheses.
- **gsub replacement:** the third argument may be a string using `%0` to `%9`, a table queried with the first capture, or a function called with the captures, and `gsub` also returns the match count.
- **Plain find:** passing `true` as the fourth argument of `string.find` turns off pattern matching and searches for the literal text.
- **Order function:** the optional second argument of `table.sort`, a function returning `true` when its first argument must come before its second, which must define a strict weak order.
- **Unstable sort:** `table.sort` may reorder elements the order function considers equal, so ties need a second key.
- **Time normalisation:** `os.time` accepts out-of-range fields such as `day 32` or `hour 1000` and returns the timestamp they imply, which makes date arithmetic easy.
- **Failure return convention:** `io` functions return a failure value plus a message and an error code instead of raising, so `io.open` is usually wrapped in `assert`.
- **Read formats:** `file:read` takes `"n"`, `"a"`, `"l"` or `"L"`, or a byte count, and `"l"`, the next line, is the default.
- **Integer and float division:** `/` always produces a float, while `//` rounds the quotient towards minus infinity and gives an integer when both operands are integers.

Summary

The standard library is a few global tables, implemented in C. String functions use 1-based, negative-friendly indices and Lua patterns, which differ from regular expressions: percent classes, quantifiers on single classes only, a lazy minus, captures, `%b` and `%f`. `gsub` takes a string, table or function replacement and returns a count. `table.sort` is

unstable and needs a strict order function. `os.time` normalises out-of-range fields. `io` functions return failure, message and code, so `io.open` is guarded with `assert`.

Using Modules in Lua

Lecture

Why this comes next

"Introduction to the Standard Library" gave you the standard library, a set of tables you get for free. This lecture is about making your own. A module is a file that builds a table and returns it, and `require` runs the file once and remembers the result.

You already know every ingredient: tables and `local`. Only `require` is new.

By the end you should be able to:

- Write a module that returns a table and keeps helpers private with `local`.
- Load it with `require` and explain why a second `require` does not rerun it.
- Describe what `package.loaded` and `package.preload` hold.
- Read a `package.path` and predict which files `require` tries, in order.
- Split a script into modules and keep globals from leaking.

1. The return-a-table pattern

Here is a statistics module, `stats.lua`. The `print` at the top is there only so you can see when the file runs.

```
LUA
-- stats.lua
print("stats.lua is running")

local M = {}                -- the public namespace

local function sum(list)    -- private: never stored in M
  local total = 0
  for _, v in ipairs(list) do total = total + v end
  return total
end

function M.mean(list) return sum(list) / #list end

function M.range(list)
  local lo, hi = math.huge, -math.huge
  for _, v in ipairs(list) do
    lo = math.min(lo, v)
    hi = math.max(hi, v)
  end
  return lo, hi
end

return M
```

The manual explains why a local helper stays private: "Lua is a lexically scoped language," and "local variables can be freely accessed by functions defined inside their scope." `sum` is a local of the file's chunk, so `M.mean` can use it, but no other file can name it. The only way in is through the table the chunk returns.

Using it:

```
LUA
local stats = require("stats")
print(stats.mean({2, 4, 9}))    --> 5.0
print(stats.range({2, 4, 9}))  --> 2 9
print(stats.sum)                --> nil
```

2. What require does

The manual spells out the steps: "The function starts by looking into the `package.loaded` table to determine whether `modname` is already loaded. If it is, then `require` returns the value stored at `package.loaded[modname]` ... Otherwise, it tries to find a loader for the module." Once a loader is found, "If the loader returns any non-nil value, `require` assigns the returned value to `package.loaded[modname]`. If the loader does not return a non-nil value and has not assigned any value to `package.loaded[modname]`, then `require` assigns `true` to this entry. In any case, `require` returns the final value of `package.loaded[modname]`."

That is why the module runs once. Here it is observed:

```
LUA
local stats = require("stats")    -- prints: stats.lua is running
local again = require("stats")    -- prints nothing
print(stats == again)             --> true
print(package.loaded["stats"] == stats) --> true
```

`package.loaded` is an ordinary table, and the standard libraries live in it too. The manual: "A table used by `require` to control which modules are already loaded. When you `require` a module `modname` and `package.loaded[modname]` is not false, `require` simply returns the value stored there." (Lua 5.4 Reference Manual, `package.loaded`.) Checked: `package.loaded.string == string` is `true`. The standalone interpreter includes all standard libraries (Lua 5.4 Reference Manual, section 7), which is why you can write `math.sin` without a `require`.

Two consequences are worth running yourself.

A module that returns nothing is stored as `true`. A file containing only `local x = 1` has no `return`, and:

```
LUA
print(require("silent"))          --> true  ./silent.lua
print(package.loaded.silent)      --> true
```

The manual gives the reason, quoted above: if the loader returns no non-nil value and has not assigned `package.loaded[modname]`, then "require assigns `true` to this entry."

You can force a reload by deleting the cache entry. Because `require` returns the stored value whenever `package.loaded[modname]` is not false, setting the entry to `nil` makes the next `require` load the file again.

```
LUA
package.loaded.stats = nil
local fresh = require("stats")    -- prints: stats.lua is running, again
print(fresh == stats)            --> false
```

Reload only in development tools, because files holding the old table now disagree with the new code.

The 5.4 manual says `require` returns "as a second result the loader data," which for a file is the file path. A module that is loaded for the first time shows it:

```
LUA
package.loaded.stats = nil
print(select(2, require("stats"))) -- the module prints its message first, then: --> ./stats.lua
```

3. package.path: how a name becomes a file

The manual: `package.path` is "A string with the path used by `require` to search for a Lua loader." Each piece of it is a template. The manual's entry for `package.searchpath` explains how a template is used: "A path is a string containing a sequence of templates separated by semicolons. For each template, the function replaces each interrogation mark (if any) in the template with a copy of name wherein all occurrences of sep (a dot, by default) were replaced by rep (the system's directory separator, by default), and then tries to open the resulting file name."

So there are two rules: the question mark is replaced by the module name, and dots in the name become directory separators. That is how `require "util.strings"` loads `util/strings.lua`.

Here is the path an interpreter built from the official 5.4.9 source starts with. Yours will differ, because the default is compiled in:

```
CODE
/usr/local/share/lua/5.4/? .lua
/usr/local/share/lua/5.4/?/init.lua
/usr/local/lib/lua/5.4/? .lua
/usr/local/lib/lua/5.4/?/init.lua
./?.lua
./?/init.lua
```

The manual says where it comes from: "At start-up, Lua initializes this variable with the value of the environment variable `LUA_PATH_5_4` or the environment variable `LUA_PATH` or with a default path defined in `luaconf.h`, if those environment variables are not defined." `package.cpath` is initialised the same way, from `LUA_CPATH_5_4` or `LUA_CPATH`.

When nothing matches, `require` lists every template it tried. Observed, trimmed:

```
CODE
module 'nothere' not found:
  no field package.preload['nothere']
  no file '/usr/local/share/lua/5.4/nothere.lua'
  no file '/usr/local/share/lua/5.4/nothere/init.lua'
  ...
```

Note the first line: before touching the disk, `require` checked `package.preload`, "A table to store loaders for specific modules." You can register a module that exists only in memory:

```
LUA
package.preload["virtual"] = function(name)
  print("loader got", name)
  return { hello = "from memory" }
end
print(require("virtual").hello)    --> loader got  virtual
                                --> from memory
```

The 5.4 manual says "require calls the loader with two arguments: modname and an extra value, a loader data." For a preload loader that extra value is the string `":preload:"`, because "The first searcher always returns the string `":preload:"`." A loader that needs only the name can ignore the second argument.

4. Hands-on: splitting a script into modules

Here is a small project. The layout is:

```
CODE
project/
  main.lua
  stats.lua
  util/
    strings.lua
```

`util/strings.lua` holds two helpers. The `trim` function uses the end anchor from "Introduction to the Standard Library", written as the escape `"\36"`:

```
LUA
-- util/strings.lua
local M = {}
local END = "\36" -- the end-of-subject anchor character

function M.trim(s)
    return (s:match("^%s*(.)%s*" .. END))
end

function M.split(s, sep) -- sep must be a single non-magic character
    local out = {}
    for piece in (s .. sep):gmatch("(.)" .. sep) do
        out[#out + 1] = piece
    end
    return out
end

return M
```

`main.lua` brings the pieces together:

```
LUA
-- main.lua
local stats = require("stats")
local strs = require("util.strings")

print("[ " .. strs.trim(" hi there ") .. "]") --> [hi there]
print(table.concat(strs.split("a,b,,c", ",", "/"))) --> a|b||c
print(stats.mean({1, 2, 3})) --> 2.0
```

Run it with `lua main.lua` from inside `project/`. Now run it from anywhere else, such as `cd /tmp; lua /path/to/project/main.lua`, and it fails with `module 'stats' not found` and the list of tried files. The first template is `./?.lua`, a relative path, so the files must be where the process is running. The fix is to put the script's own folder at the front of the path, using the script name that the standalone interpreter stores in `arg[0]` (Lua 5.4 Reference Manual, section 7):

```
LUA
local here = arg[0]:match("^(.*)/[/]*") or "."
package.path = here .. "/?.lua;" .. package.path
```

With those two lines at the top of `main.lua`, the same run from another directory succeeds. The manual's wording backs the move: "A path is a string containing a sequence of templates separated by semicolons," so putting a new template first is just string concatenation. The `arg[0]` trick relies on the standalone `lua` program, which is the one that fills the `arg` table.

5. Keeping globals out

A module's biggest hazard is a forgotten `local`. Here is a module with that bug:

```
LUA
-- leaky.lua
local M = {}
helper = function() return "oops, a global" end -- forgot local
function M.run() return helper() end
return M
```

After `require("leaky")`, observed: `helper` is a global visible to every other file. Two files that both forget `local` on a name `helper` will silently overwrite each other. The manual explains the mechanism: "any reference to a free name (that is, a name not bound to any declaration) var is syntactically translated to `_ENV.var`," so a global is just a

field of the environment table, and "An assignment to a global name `x = val` is equivalent to the assignment `_ENV.x = val`."

One cure is a metatable on the global table, with Chapter 5's `__newindex`, which turns every assignment to an undeclared global into an error:

```
LUA
setmetatable(_G, { __newindex = function(_, name)
    error("assignment to undeclared global '" .. name .. "'", 2)
end })
print(pcall(require, "Leaky"))
--> false ./leaky.lua:3: assignment to undeclared global 'helper'
```

Put that guard in your test runner only.

6. What to take away

A module is a file that builds a table and returns it. `local` keeps helpers private, and the returned table is the only public surface. `require` checks `package.loaded`, then `package.preload`, then searches `package.path` template by template, replacing the question mark with the name and dots with directory separators. It runs the file once, stores what the file returned (or `true` if nothing) in `package.loaded`, and hands back the same value on every later call. The default path starts at the relative `./?.lua`, so prepend your script's folder. A forgotten `local` creates a shared global, so guard against that in tests.

Next chapter: coroutines, Lua's cooperative multitasking.

Key Terms

- **Module:** a file whose chunk builds a table of functions and constants and returns it, so the table acts as a namespace for everything the module exports.
- **Return-a-table pattern:** the basic way to write a module, creating a local table `M`, adding public functions to it and ending the file with `return M`.
- **Private helper:** a function declared local in the module's chunk, visible to the module's own functions through lexical scoping but not exported.
- **require:** the function that loads a module by name, running its file once and returning the value stored in `package.loaded` for that name.
- **package.loaded:** the table `require` consults first, mapping module names to the values their loaders returned, which also holds the standard libraries.
- **Module caching:** because `require` returns the `package.loaded` entry on every later call, a module's code runs only once however many files require it.
- **Implicit true result:** if a module file returns nothing and sets no `package.loaded` entry, `require` stores `true` so the module is not run again.
- **Forced reload:** setting `package.loaded[name]` to `nil` makes the next `require` run the file again and return a new table.
- **package.path:** the string of semicolon separated templates `require` searches for Lua files, with each question mark replaced by the module name.
- **Dotted module name:** a name such as `util.strings` whose dot becomes a directory separator when searching, so it loads `util/strings.lua`.
- **Search order:** `require` checks `package.loaded`, then `package.preload`, then each `package.path` template in turn, and its error lists every location it tried.
- **package.preload:** a table mapping module names to loader functions, letting a module exist only in memory.

- **Loader data:** in Lua 5.4 `require` passes the loader a second argument and returns it as a second result, which for a file is the file path.
- **Global leak:** assigning without `local` inside a module creates a global shared by every file, which a `__newindex` guard on the global table can catch.

Summary

A module is a file that builds a table and returns it, with `local` keeping helpers private. `require` checks `package.loaded`, then `package.preload`, then each `package.path` template, replacing the question mark with the name and dots with directory separators. It runs the file once and stores the returned value, or `true` if there was none, so later calls return the same table. The default path starts at `./?.lua`, relative to where the process runs, so prepend the script's folder to run from elsewhere. A forgotten `local` leaks a global.

Chapter 7: Coroutines

Introduction to Coroutines

Lecture

Why this comes next

Chapter 4 gave you the generic `for` and the idea that an iterator is a function that remembers where it was. Writing one by hand meant packing that memory into closures or tables. A coroutine removes the packing: the function simply stops in the middle, keeps its locals and its place in the code, and later continues from exactly there. If you write Python, this is the feeling of a generator, in a more general form. This lecture covers the mechanics: the functions that drive a coroutine, its four states, how values travel in and out, and what happens when something fails.

One idea comes first: **a coroutine is not a thread**. Nothing runs in parallel, nothing is interrupted, and nothing switches unless your code asks.

By the end you should be able to:

- Explain why a coroutine is cooperative, and not an operating-system thread.
- Create a coroutine, start it, suspend it, and read its status.
- Name the four states and say which call causes each transition.
- Pass values in both directions across `resume` and `yield`.
- Explain the difference between `coroutine.resume` and `coroutine.wrap`, especially for errors.
- Say what `isyieldable`, `running`, and `close` do.

1. What a coroutine is

The manual opens its coroutine section with the definition:

"Lua supports coroutines, also called *collaborative multithreading*. A coroutine in Lua represents an independent thread of execution. Unlike threads in multithread systems, however, a coroutine only suspends its execution by explicitly calling a `yield` function." (*Lua 5.4 Reference Manual*, section 2.6)

Programming in Lua spells out what "independent" means and where the independence stops:

"A coroutine is similar to a thread (in the sense of multithreading): a line of execution, with its own stack, its own local variables, and its own instruction pointer; but sharing global variables and mostly anything else with other coroutines." (*Programming in Lua*, 1st ed., ch. 9)

So it owns its stack and locals and shares the globals. Now the sentence that separates it from a thread:

"The main difference between threads and coroutines is that, conceptually (or literally, in a multiprocessor machine), a program with threads runs several threads concurrently. Coroutines, on the other hand, are collaborative: A program with coroutines is, at any given time, running only one of its coroutines and this running coroutine only suspends its execution when it explicitly requests to be suspended." (*Programming in Lua*, 1st ed., ch. 9)

The manual also settles what the word "thread" means here. Lua's type is called `thread`, but "Lua threads are not related to operating-system threads" (*Lua 5.4 Reference Manual*, section 2.1, which adds that Lua supports coroutines "on all systems, even those that do not support threads natively").

Three consequences follow:

1. **Only one coroutine runs at a time**, so more of them is no speedup.
2. **Nobody can stop a running coroutine from outside.** "While a coroutine is running, it cannot be stopped from the outside. It only suspends execution when it explicitly requests so (through a call to `yield`)."
(PiL 1st ed., §9.4)
3. **No switch can land in the middle of your code.** The same passage calls this an advantage: "all synchronization among threads is explicit in the program."

For Python readers, `next(gen)` is roughly the Lua `resume`. The book draws the line precisely (PiL 1st ed., §9.1). Lua offers "what I call asymmetric coroutines. That means that it has a function to suspend the execution of a coroutine and a different function to resume a suspended coroutine," as opposed to symmetric coroutines, "where there is only one function to transfer control from any coroutine to another." Lua's are also what the book calls "true" coroutines, and "Using Coroutines for Concurrency" shows where that beats a generator.

2. Create, resume, yield

Two library calls start a coroutine's life, and the manual is careful about the order:

"Creates a new coroutine, with body `f`. `f` must be a function. Returns this new coroutine, an object with type "thread"." (`coroutine.create`)

Creating does not run anything: `create` "does not start the coroutine" (section 2.6). Running starts with `coroutine.resume`:

"Starts or continues the execution of coroutine `co`. The first time you resume a coroutine, it starts running its body. The values `val1, ...` are passed as the arguments to the body function. If the coroutine has yielded, resume restarts it; the values `val1, ...` are passed as the results from the yield."
(`coroutine.resume`)

Here is the whole idea at its smallest:

```
LUA
local co = coroutine.create(function() print("body started") end)
print(type(co), coroutine.status(co))
print("nothing printed yet")
print(coroutine.resume(co))
print(coroutine.status(co))
```

The output:

```
CODE
thread suspended
nothing printed yet
body started
true
dead
```

The body ran only at the `resume`, and the coroutine is `dead` because the body returned. `true` is the "no error" flag.

The other half is `coroutine.yield`: "Suspends the execution of the calling coroutine. Any arguments to yield are passed as extra results to resume." The manual's own worked example exercises everything at once. Read it slowly, then run it:

```
LUA
function foo (a)
  print("foo", a)
  return coroutine.yield(2*a)
end
```

```

co = coroutine.create(function (a,b)
    print("co-body", a, b)
    local r = foo(a+1)
    print("co-body", r)
    local r, s = coroutine.yield(a+b, a-b)
    print("co-body", r, s)
    return b, "end"
end)

print("main", coroutine.resume(co, 1, 10))
print("main", coroutine.resume(co, "r"))
print("main", coroutine.resume(co, "x", "y"))
print("main", coroutine.resume(co, "x", "y"))

```

```

CODE
co-body 1 10
foo 2
main true 4
co-body r
main true 11 -9
co-body x y
main true 10 end
main false cannot resume dead coroutine

```

This matches the output the manual prints for the same code. Trace the first `resume`: the body starts with `a=1`, `b=10`, calls `foo(2)`, which prints and yields `4`, and that `4` comes out of `resume` as the second result. The second `resume` sends `"r"` in, and it comes out of the `yield` inside `foo`, so the body prints `co-body r`. On the last line, resuming the finished coroutine returns `false` plus a message rather than crashing the program.

3. The four states

A coroutine is in one of four states. The manual defines what `coroutine.status` returns:

"Returns the status of the coroutine `co`, as a string: "running", if the coroutine is running (that is, it is the one that called status); "suspended", if the coroutine is suspended in a call to yield, or if it has not started running yet; "normal" if the coroutine is active but not running (that is, it has resumed another coroutine); and "dead" if the coroutine has finished its body function, or if it has stopped with an error." (`coroutine.status`)

A coroutine never resumed is `suspended`, and `dead` covers both a normal finish and a failure.

The `normal` state appears when coroutines resume each other: the manual's definition is "active but not running (that is, it has resumed another coroutine)." This program makes all four visible:

```

LUA
local outer, inner
inner = coroutine.create(function()
    print("inner sees outer as:", coroutine.status(outer))
    print("inner sees itself as:", coroutine.status(inner))
    coroutine.yield()
end)
outer = coroutine.create(function()
    print("outer sees itself as:", coroutine.status(outer))
    coroutine.resume(inner)
    print("outer sees inner as:", coroutine.status(inner))
end)
print("before:", coroutine.status(outer))
coroutine.resume(outer)
print("after:", coroutine.status(outer), coroutine.status(inner))

```

```

CODE
before: suspended
outer sees itself as:  running
inner sees outer as:   normal
inner sees itself as:  running
outer sees inner as:   suspended
after:  dead    suspended

```

While `inner` runs, `outer` is `normal`: active, but waiting for `inner` to hand control back. When `inner` yields, `outer` is `running` again, and at the end `outer` is `dead` while `inner` is parked at its yield.

The transitions:

- `suspended` to `running`: someone calls `resume` on it.
- `running` to `suspended`: it calls `yield`.
- `running` to `normal`: it calls `resume` on another coroutine.
- `normal` to `running`: that other coroutine yields or finishes.
- `running` to `dead`: the body returns, or an error escapes the body.

4. Passing values in both directions

The book's phrase for this (PiL 1st ed., §9.1) is "a pair resume-yield can exchange data," and there are exactly four channels. Each one is shown with the book's own example.

```

LUA
local co = coroutine.create(function (a, b, c)
    print("co", a, b, c + 2)
end)
coroutine.resume(co, 1, 2, 3)

co = coroutine.create(function (a, b)
    coroutine.yield(a + b, a - b)
end)
print(coroutine.resume(co, 20, 10))

co = coroutine.create(function (x)
    print("co1", x)
    print("co2", coroutine.yield())
end)
coroutine.resume(co, "hi")
coroutine.resume(co, 4, 5)

co = coroutine.create(function () return 6, 7 end)
print(coroutine.resume(co))

```

```

CODE
co 1    2    5
true   30   10
co1 hi
co2 4    5
true   6    7

```

In order:

1. **Into the body.** The first `resume` "passes its extra arguments as arguments to the coroutine main function," because no `yield` is waiting yet.
2. **Out through `yield`.** `resume` returns, "after the true that signals no errors, any arguments passed to the corresponding yield."
3. **Back in through `yield`.** "Symmetrically, yield returns any extra arguments passed to the corresponding resume." The value you pass to the *next* resume is what the *previous* yield evaluates to.
4. **Out through return.** Finally, "when a coroutine ends, any values returned by its main function go to the corresponding resume."

Think of `resume` and `yield` as two ends of one conversation: each call hands over its arguments and receives the other side's as its results.

Yielding from deep inside. The manual states that a yield does not have to be in the body itself: "the corresponding `coroutine.resume` returns immediately, even if the yield happens inside nested function calls (that is, not in the main function, but in a function directly or indirectly called by the main function)."

```
LUA
local function walk(depth)
  if depth == 0 then
    coroutine.yield("reached the bottom")
  else
    walk(depth - 1)
  end
  return "unwound"
end
local co = coroutine.create(function() return walk(3) end)
print(coroutine.resume(co))
print(coroutine.resume(co))
print(coroutine.resume(co))
print(coroutine.resume(co))
```

```
CODE
true   reached the bottom
true   unwound
dead
```

Four frames deep, the yield still suspends the whole coroutine. PiL 1st ed. calls these "true" coroutines and says that in the restricted kind, which it ties to Python generators, "only the main body of such semi-coroutines can yield" (§9.1).

5. resume versus wrap, and what errors do

There are two ways to drive a coroutine, and they treat failure differently. Start with the manual's description of what happens when a body fails:

"In case of errors, `coroutine.resume` returns false plus the error object. In this case, the coroutine does not unwind its stack, so that it is possible to inspect it after the error with the debug API." (*Lua 5.4 Reference Manual*, section 2.6)

PiL 1st ed. adds: "Note that resume runs in protected mode. Therefore, if there is any error inside a coroutine, Lua will not show the error message, but instead will return it to the resume call." (§9.1)

`coroutine.wrap` packages `create` and `resume` into a single function:

"Creates a new coroutine, with body `f`; `f` must be a function. Returns a function that resumes the coroutine each time it is called. Any arguments passed to this function behave as the extra

arguments to resume. The function returns the same values returned by resume, except the first boolean. In case of error, the function closes the coroutine and propagates the error."

(coroutine.wrap)

PiL 1st ed. words the same difference briefly: that function "does not return an error code as its first result; instead, it raises the error in case of errors." (§9.3) Side by side, saved as `demo.lua`:

```
LUA
local function boom() error("something broke") end

local co = coroutine.create(boom)
print("resume:", coroutine.resume(co))
print("status:", coroutine.status(co))
print("again :", coroutine.resume(co))

local f = coroutine.wrap(boom)
print("wrap  :", pcall(f))
print("wrap again:", pcall(f))
```

```
CODE
resume: false    demo.lua:1: something broke
status: dead
again : false    cannot resume dead coroutine
wrap  : false    demo.lua:1: something broke
wrap again: false cannot resume dead coroutine
```

The `pcall` around the wrapped function catches the raised error; without it the program would stop at that line. Use `resume` when you want to handle failure yourself, such as a scheduler that must survive one dying task ("Using Coroutines for Concurrency"). Use `wrap` when the coroutine is an implementation detail, such as an iterator in a `for` loop, and failure should look like any failing call. After an error the coroutine is `dead`; retrying means creating a new one.

A `yield` inside `pcall` inside a coroutine is allowed in 5.4. The coroutine is suspended inside the protected call and carries on after it when resumed:

```
LUA
local co = coroutine.create(function()
  local ok, err = pcall(function() coroutine.yield(1) end)
  print("pcall returned", ok, err)
end)
print(coroutine.resume(co))
print(coroutine.resume(co))
```

```
CODE
true 1
pcall returned true nil
true
```

The manual's C API section explains why this is possible: "Lua raises an error whenever it tries to yield across an API call, except for three functions: `lua_yieldk`, `lua_callk`, and `lua_pcallk`" (section 4.5, "Handling Yields in C"). The last of those, the manual says, "behaves exactly like `lua_pcall`, except that it allows the called function to yield" (Lua 5.4 Reference Manual, `lua_pcallk`).

6. isyieldable, running and close

Three more functions complete the library. Here they are on Lua 5.4; the address printed by `coroutine.running` will differ on your machine:

```
LUA
print(coroutine.isyieldable())    --> false, the main thread cannot yield
print(coroutine.running())        --> thread: 0x... true
print(coroutine.wrap(function() return coroutine.isyieldable() end)())    --> true

local co = coroutine.create(function() coroutine.yield() end)
coroutine.resume(co)
print(coroutine.status(co))       --> suspended
print(coroutine.close(co))        --> true
print(coroutine.status(co))       --> dead
```

The manual's wording for each one:

"Returns true when the coroutine `co` can yield. The default for `co` is the running coroutine. A coroutine is yieldable if it is not the main thread and it is not inside a non-yieldable C function."

(`coroutine.isyieldable`)

A helper that may be called from either context can check it before deciding whether to yield.

"Returns the running coroutine plus a boolean, true when the running coroutine is the main one."

(`coroutine.running`)

The second result is how code detects the main thread.

"Closes coroutine `co`, that is, closes all its pending to-be-closed variables and puts the coroutine in a dead state. The given coroutine must be dead or suspended. In case of error (either the original error that stopped the coroutine or errors in closing methods), returns false plus the error object; otherwise returns true." (`coroutine.close`)

The manual explains why: "If a coroutine yields and is never resumed again, some variables may never go out of scope, and therefore they will never be closed," and a coroutine that ends with an error "does not unwind its stack, so it does not close any variable." So `close` is the tool for abandoning a coroutine, such as a generator you stopped consuming early.

7. What to take away

- A coroutine has its own stack and locals, pauses only at its own `yield` calls, and only one runs at a time.
- `create` makes it, `resume` runs it, `yield` pauses it, and `status` reports `suspended`, `running`, `normal`, or `dead`.
- Values travel through four channels: resume arguments in, yield arguments out, resume arguments back in through yield, and return values out.
- A yield can happen at any call depth below the body.
- `resume` returns `false` plus the error and leaves the coroutine `dead`. `wrap` raises the error and closes the coroutine.

Next: iterators, pipelines, and a scheduler, each only a few lines.

Key Terms

- **Coroutine:** an independent line of execution with its own stack and locals that shares globals with other coroutines and suspends only when it explicitly calls `yield`.

- **Cooperative scheduling:** switching between coroutines only at yield calls the programmer wrote, so a running coroutine cannot be interrupted from outside.
- **Lua thread:** the type named thread, which represents a coroutine and is not related to operating-system threads.
- **coroutine.create:** builds a new coroutine from a function body and returns a thread value without starting it.
- **coroutine.resume:** starts or continues a coroutine, passing its extra arguments to the body or to the pending yield, and returns true or false followed by values or the error.
- **coroutine.yield:** suspends the calling coroutine and passes its arguments out as extra results of the matching resume.
- **Coroutine status:** the string returned by coroutine.status, one of suspended, running, normal or dead.
- **Normal state:** the status of a coroutine that is active but not running because it has resumed another coroutine.
- **Dead coroutine:** one whose body has returned or stopped with an error, which cannot be resumed again and makes resume return false with the message cannot resume dead coroutine.
- **resume-yield data exchange:** arguments to the first resume become body parameters, yield arguments come out of resume, later resume arguments come out of yield, and body return values come out of the last resume.
- **Yield from nested calls:** a yield may occur in any function called directly or indirectly by the coroutine body and still suspends the whole coroutine.
- **coroutine.wrap:** creates a coroutine and returns a function that resumes it, dropping the leading boolean and raising errors instead of returning them.
- **Asymmetric coroutines:** coroutines with one function to suspend and a different function to resume, which is how Lua works, in contrast to symmetric coroutines with a single transfer function.
- **coroutine.isyieldable, coroutine.running and coroutine.close:** functions that report whether a coroutine can yield, return the running coroutine plus a boolean that is true for the main one, and close a suspended or dead coroutine together with its pending to-be-closed variables.

Summary

A coroutine is a function with its own stack and locals that pauses only at its own yield calls, and only one runs at a time, so it is cooperative rather than pre-emptive and is not an operating-system thread. create builds it, resume runs it, yield pauses it, and status reports suspended, running, normal or dead. Values flow in and out through the arguments and results of resume and yield, and a yield can occur at any call depth. resume returns false plus the error and leaves the coroutine dead, while wrap raises the error and closes the coroutine. isyieldable reports whether a coroutine can yield, running also says whether it is the main thread, and close abandons a suspended coroutine cleanly.

Using Coroutines for Concurrency

Lecture

Why this comes last

"Introduction to Coroutines" gave you the mechanics. This lecture puts them to work in three shapes: an iterator that feeds a generic `for`, a pipeline of producers and consumers, and a scheduler that interleaves several tasks. It ends on the part that is easy to skip and expensive to forget: what cooperative scheduling cannot do.

By the end you should be able to:

- Write a generator with `coroutine.wrap` and use it in a generic `for`.

- Explain why a recursive generator needs no extra machinery in Lua.
- Build a producer, filter, and consumer pipeline with `resume` and `yield`.
- Write a round-robin scheduler that removes finished tasks safely.
- State the honest limits: no parallelism, and a task that never yields stalls everything.

1. Iterators are producers

Chapter 4's generic `for` calls an iterator function until it returns `nil`. The book observes that this is a special case of something bigger:

"We can see loop iterators as a quite specific example of the producer-consumer pattern. An iterator produces items to be consumed by the loop body. Therefore, it seems appropriate to use coroutines to write iterators. Actually, coroutines provide a powerful tool for this task."

(*Programming in Lua*, 1st ed., §9.3, "Coroutines as Iterators")

The pattern has a name in the book: "packs a call to resume with its corresponding coroutine inside a function. This pattern is so common that Lua provides a special function for it: `coroutine.wrap`." The wrapped function returns the yielded values, and, per the manual, "returns the same values returned by `resume`, except the first boolean." So a counter becomes:

```
LUA
local function range(n)
  return coroutine.wrap(function()
    for i = 1, n do coroutine.yield(i) end
  end)
end
for i in range(4) do io.write(i, " ") end
print()
```

```
CODE
1 2 3 4
```

Compare it with the closure version from Chapter 4. There, the loop variable and position had to be stored in upvalues and updated by hand on every call. Here the `for` inside the body is the state. When the body finishes, the wrapped function returns nothing, which is `nil` to the generic `for`, and the loop ends. In Python terms this is `yield` inside a generator function, and `range` here is a generator factory.

The recursion that shows what "true" means

The book's showpiece is a permutation generator. It starts with an ordinary recursive function that prints each result, and the only change is replacing the print with a yield:

```
LUA
local function printResult(a) print(table.concat(a, " ")) end
local 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
local function permutations(a)
  return coroutine.wrap(function() permgen(a) end)
```

```
end
for p in permutations{"a", "b", "c"} do printResult(p) end
```

CODE

```
b c a
c b a
c a b
a c b
b a c
a b c
```

The yield sits several recursive calls below the coroutine body, and the manual guarantees that works: "the corresponding coroutine.resume returns immediately, even if the yield happens inside nested function calls (that is, not in the main function, but in a function directly or indirectly called by the main function)." *Programming in Lua* is explicit about why this matters for Python readers. It calls a restricted coroutine, where only the main body can yield, a semi-coroutine, and says "A generator in Python is an example of this meaning of semi-coroutines." Then: "the difference between coroutines and generators (as presented in Python) is a deep one; generators are simply not powerful enough to implement several interesting constructions that we can write with true coroutines. Lua offers true, asymmetric coroutines." (PiL 1st ed., §9.1)

The practical reading: to turn a recursive tree walk into a Lua iterator you change one line. You do not restructure the recursion into an explicit stack.

2. Producers and consumers

The book frames the problem with two functions, each with its own loop:

```
LUA
function producer ()
  while true do
    local x = io.read()    -- produce new value
    send(x)                -- send it to consumer
  end
end

function consumer ()
  while true do
    local x = receive()    -- receive value from producer
    io.write(x, "\n")      -- consume it
  end
end
```

The difficulty it names: "It is a typical case of a who-has-the-main-loop problem. Both the producer and the consumer are active, both have their own main loops, and both assume that the other is a callable service." Without coroutines you would have to restructure one side by unrolling its loop and making it a passive agent (§9.2).

The coroutine answer, quoted from the book: "a resume-yield pair turns upside-down the typical relationship between caller and callee. When a coroutine calls yield, it does not enter into a new function; instead, it returns a pending call (to resume)." Both sides keep their loops. `receive` is a `resume`, `send` is a `yield`:

```
LUA
function receive (prod)
  local status, value = coroutine.resume(prod)
  return value
end

function send (x)
  coroutine.yield(x)
end
```

The book then adds a **filter**, "a consumer and a producer at the same time, so it resumes a producer to get new values and yields the transformed values to a consumer." Its version reads from `io.read`. This one reads from a list so the output is repeatable, and stops cleanly when the producer is finished:

```
LUA
local lines = {"alpha", "beta", "gamma"}

local function receive(prod)
    local status, value = coroutine.resume(prod)
    return value
end
local function send(x) coroutine.yield(x) end

local function producer()
    return coroutine.create(function()
        for _, x in ipairs(lines) do send(x) end
        end)
end

local function filter(prod)
    return coroutine.create(function()
        for line = 1, math.huge do
            local x = receive(prod)
            if x == nil then return end -- upstream finished: finish too
            send(string.format("%5d %s", line, x))
        end
    end)
end

local function consumer(prod)
    while true do
        local x = receive(prod)
        if x == nil then break end
        io.write(x, "\n")
    end
end

consumer(filter(producer()))
```

```
CODE
1 alpha
2 beta
3 gamma
```

Two things differ from the book's version, and both are worth noting. The book's loops are `while true` with no end test, so ours add a check for `nil` to stop when the producer finishes. And `receive` ignores the `status` result. That is fine here, but if the producer raised an error, `resume` would return `false` plus the message, and `receive` would hand that message downstream as if it were data. With a producer whose body is `error("disk full")`, the first `receive` returns the message with the file name and line number in front of it, and every later `receive` returns `cannot resume dead coroutine`, which is never `nil`, so a consumer loop that waits for `nil` would never stop. Section 4 returns to this.

The book names the shape of this design: "the program starts calling the consumer. When the consumer needs an item, it resumes the producer, which runs until it has an item to give to the consumer, and then stops until the consumer restarts it again. Therefore, we have what we call a consumer-driven design." (PiL 1st ed., §9.2)

3. A round-robin scheduler

The next step is many coroutines, each a task, with a loop that gives each a turn. The book's version drives downloads with `LuaSocket`, and its description of the dispatcher is worth keeping: "It is mainly a loop that goes

through all threads, calling one by one. It must also remove from the list the threads that finish their tasks. It stops the loop when there are no more threads to run." (PiL 1st ed., §9.4)

In the book's version a task yields a non-`false` value while it still has work ("The non-`false` argument passed to `yield` signals to the dispatcher that the thread is still performing its task."), and the dispatcher drops a thread when its resume gives no such value (§9.4). This scheduler checks `coroutine.status` instead and uses `resume`, so one failing task cannot stop the rest. It is runnable anywhere:

```
LUA
local tasks = {}

local function spawn(name, steps)
    local co = coroutine.create(function()
        for i = 1, steps do
            print(name .. " step " .. i)
            coroutine.yield()          -- hand control back to the scheduler
        end
    end)
    table.insert(tasks, co)
end

local function run()
    local i = 1
    while #tasks > 0 do
        if i > #tasks then i = 1 end
        local co = tasks[i]
        local ok, err = coroutine.resume(co)
        if not ok then print("task failed: " .. tostring(err)) end
        if coroutine.status(co) == "dead" then
            table.remove(tasks, i)    -- do not advance; the next task slid into slot i
        else
            i = i + 1
        end
    end
end

spawn("A", 3)
spawn("B", 1)
spawn("C", 2)
run()
print("all tasks finished")
```

```
CODE
A step 1
B step 1
C step 1
A step 2
C step 2
A step 3
all tasks finished
```

Read the interleaving. Each task prints, yields, and waits for its next turn. B finishes after one step and is removed, so from the second round only A and C alternate, and A runs on alone at the end. Nothing here is parallel: the order is fully determined by the list and by where each task chose to yield.

The `table.remove` line has the classic list-removal trap. Removing slot `i` slides the next task into slot `i`, so the loop must not advance `i` in that case. PiL 1st ed. avoids the problem by leaving its inner loop right after a removal (§9.4).

4. Errors inside a scheduled task

"Introduction to Coroutines" showed that `resume` returns `false` plus the message and leaves the coroutine `dead`. That is exactly why the scheduler above uses `resume` and not `wrap`. With `wrap`, the manual says the function "closes the coroutine and propagates the error," which would unwind into `run` and take every other task down with it

unless you wrapped each call in `pcall`. A scheduler that should survive a bad task wants the plain `false` result, which is the value the `receive` helper above threw away. In your own code, check the first result of `resume` before trusting the second.

5. The honest limits

Everything so far depends on one fact from "Introduction to Coroutines": the running coroutine yields only when it chooses. The book states the cost directly:

"However, with non-preemptive multithreading, whenever any thread calls a blocking operation, the whole program blocks until the operation completes." (Programming in Lua, 1st ed., §9.4)

Here is the failure in miniature. A task that computes without ever yielding holds the only turn until it is done, so a polite task behind it waits:

```
LUA
local order = {}
local hog = coroutine.create(function()
    table.insert(order, "hog starts")
    local n = 0
    for i = 1, 5e6 do n = n + i end      -- pure computation, no yield
    table.insert(order, "hog ends")
end)
local polite = coroutine.create(function()
    for i = 1, 2 do
        table.insert(order, "polite " .. i)
        coroutine.yield()
    end
end)
for _, co in ipairs{hog, polite, hog, polite} do coroutine.resume(co) end
print(table.concat(order, " / "))
```

```
CODE
hog starts | hog ends | polite 1 | polite 2
```

The hog finished its whole loop before the polite task ran once. Scale that loop up to a real blocking call, such as reading a socket, and the entire program freezes for its duration. The book's remedy is to make the blocking call cooperative. Its socket `receive` sets a zero timeout and yields instead of waiting: "if there is not enough data available, it yields." The scheduler then runs other tasks in the meantime.

The book is equally frank about the next cost. When no task has data, the simple dispatcher keeps cycling: "the dispatcher does a busy wait, going from thread to thread only to check that they still have no data. As a result, this coroutine implementation uses almost 30 times more CPU than the sequential solution." (PiL 1st ed., §9.4) Its fix is to block on all the waiting sockets at once with `select`, which this lecture does not run because it needs `LuaSocket`.

Collect the limits into three plain statements:

1. **No parallelism.** "A program with coroutines is, at any given time, running only one of its coroutines." Coroutines help when tasks spend their time waiting, and do nothing for a task that is busy computing.
2. **One task that never yields stalls everything,** as the hog showed.
3. **Waiting must be cooperative.** A scheduler needs a way to know a task is not ready, either a yielded status or a readiness check like `select`, or it burns CPU polling.

The upside the book gives is real: "Programming is much easier in the absence of preemption. You do not need to be paranoid about synchronization bugs, because all synchronization among threads is explicit in the program. You only have to ensure that a coroutine only yields when it is outside a critical region." (PiL 1st ed., §9.4)

6. What to take away

- A generator is `coroutine.wrap` around a body that yields. A recursive body works unchanged, because Lua offers true coroutines, not the restricted kind that Programming in Lua ties to Python generators.
- A producer, filter, and consumer pipeline is `resume` as `receive` and `yield` as `send`, with each side keeping its own loop.
- A round-robin scheduler is a list of coroutines, a loop that resumes each, and removal of the dead ones without skipping a task.
- Use `resume` in a scheduler so one failing task cannot end the program, and always read its first result.
- Coroutines give you cooperation, not parallelism. A task that computes or blocks without yielding stalls every other task.

Chapter 8 moves to the host side of Lua, embedding it in a program.

Key Terms

- **Coroutine generator:** a `coroutine.wrap` function whose body yields each value in turn, so that it can drive a generic for loop.
- **Coroutines as iterators:** the book's observation that a loop iterator is a producer and the loop body its consumer, which makes coroutines a natural way to write iterators.
- **True coroutines:** coroutines that can yield from any call depth, unlike the restricted semi-coroutines that Programming in Lua ties to Python generators, where only the main body can yield.
- **Producer-consumer:** two cooperating routines where one generates values and the other uses them, matched with a coroutine so each keeps its own main loop.
- **Who has the main loop:** the book's name for the problem of which side of a producer and consumer owns the main loop, solved because a resume-yield pair turns the relationship between caller and callee upside-down.
- **Filter:** a coroutine that sits between a producer and a consumer, resuming the producer for values and yielding transformed values onward.
- **Consumer-driven design:** a pipeline where the program starts by calling the consumer, which resumes the producer whenever it needs an item.
- **Round-robin scheduler:** a loop over a list of coroutines that resumes each in turn and removes the dead ones, giving every task a turn.
- **Safe removal:** when `table.remove` deletes slot `i` the next task slides into slot `i`, so the scheduler must not advance its index in that case.
- **Scheduler error handling:** a scheduler should use `resume` rather than `wrap` so a failing task returns false and a message instead of propagating an error through every other task.
- **Non-preemptive multithreading:** switching only at explicit yields, which makes synchronization explicit and removes most races but lets any blocking call stall the whole program.
- **Starvation:** a task that computes or blocks without yielding holds the only running turn, so every other task waits until it finishes.
- **Busy wait:** a dispatcher that keeps cycling through tasks with no data, which the book reports as using almost 30 times more CPU than the sequential solution.

Summary

A generator is `coroutine.wrap` around a body that yields, and a recursive body works unchanged because Lua coroutines are true coroutines, not the restricted kind tied to Python generators. A producer, filter and consumer pipeline uses `resume` as `receive` and `yield` as `send`, with each side keeping its own loop. A round-robin scheduler is a list of coroutines resumed in turn, with finished ones removed without skipping a task, and it should use `resume`

so one failing task cannot stop the rest. Coroutines give cooperation, not parallelism: only one runs at a time, a task that never yields stalls everything, and waiting must be cooperative or the scheduler busy-waits.

Chapter 8: Embedding and Synthesis

Embedding Lua in a Host Environment

Lecture

Why this comes first

For seven chapters Lua has been the thing you write. This chapter you meet the thing that runs it. The manual opens with the fact that explains the whole book: "As an extension language, Lua has no notion of a 'main' program: it works *embedded* in a host client, called the *embedding program* or simply the *host*." Even the `lua` program you have been running is a host: "Lua code starts running by a call from C code in the host program. (When you use Lua standalone, the lua application is the host program.)"

Seen from the host side, the odd choices stop being odd. A C program must hand values in, take values out, survive errors, and restrict what a script can touch, all through a small interface. Every program below was compiled and run against the Lua 5.4 headers and library; where a listing shows only the interesting lines, the rest is the setup from Section 1.

By the end you should be able to:

- Say what a `lua_State` is and how `luaL_newstate` differs from `lua_newstate`.
- Embed Lua in C: create a state, open libraries, load a chunk, run it protected, close the state.
- Read and write the virtual stack with positive and negative indices.
- Expose a C function to Lua, and call a Lua function from C with `lua_pcall`.
- Restrict a script's environment with a custom `_ENV` table, from Lua with `load` and from C with `lua_setupvalue`.

1. The smallest embedder

```
C
#include <stdio.h>
#include "lua.h"
#include "lauxlib.h"
#include "lualib.h"

int main(void) {
    lua_State *L = luaL_newstate();      /* create a state with default allocator */
    luaL_openlibs(L);                   /* make print, string, table, math... visible */

    /* luaL_loadstring only COMPILES; lua_pcall RUNS, in protected mode.
       The || works because both return 0 on success. */
    int status = luaL_loadstring(L, "print('hello from', _VERSION)")
        || lua_pcall(L, 0, 0, 0);
    if (status != 0) {
        fprintf(stderr, "error: %s\n", lua_tostring(L, -1));
        lua_pop(L, 1);                  /* the error object is on the stack: pop it */
    }
    lua_close(L);                       /* release the whole state */
    return status;
}
```

Built against the 5.4 headers and library, for example with `gcc hello.c -o hello -I lua-5.4.9/src lua-5.4.9/src/liblua.a -lm -ldl` from a source tree where `make` has run, it prints (a tab separates the fields):

```
CODE
hello from  Lua 5.4
```

`luaL_openlibs` "opens all standard Lua libraries into the given state." A fresh state has no `print` and no `string`; *PiL 1st ed.* explains that "To keep Lua small, all standard libraries are provided as separate packages, so that you do not have to use them if you do not need to." (§24.1) `luaL_loadstring` is careful about its limit: "this function only loads the chunk; it does not run it." `lua_close` will "release all objects in the given Lua state ... and frees all dynamic memory used by this state."

2. What a lua_State is

The manual: `lua_State` is "an opaque structure that points to a thread and indirectly (through the thread) to the whole state of a Lua interpreter. The Lua library is fully reentrant: it has no global variables." Two consequences.

There is no global interpreter. Because the library has no global variables, a host can create as many states as it likes, and each one is a separate interpreter. A host can run many scripts in many states.

The type is really a thread. "The type `lua_State` (despite its name) refers to a thread," and "each Lua state has one or more threads, which correspond to independent, cooperative lines of execution." That is Chapter 7's coroutine seen from C.

`luaL_newstate` versus `lua_newstate`. They differ in who picks the allocator. `lua_newstate (lua_Alloc f, void *ud)` takes "the allocator function; Lua will do all memory allocation for this state through this function."

`luaL_newstate` takes nothing: it "calls `lua_newstate` with an allocator based on the ISO C allocation functions and then sets a warning function and a panic function ... that print messages to the standard error output." A counting allocator shows the hook:

```
C
static void *counting_alloc(void *ud, void *ptr, size_t osize, size_t nsize) {
    (void)ud;
    calls++;
    if (ptr) live_bytes -= osize;
    if (nsize == 0) { free(ptr); return NULL; } /* nsize 0 means free */
    live_bytes += nsize;
    return realloc(ptr, nsize);
}
/* lua_State *L = luaL_newstate(counting_alloc, NULL); then lua_close(L); */
```

```
CODE
fresh state: 55 allocator calls, 4987 live bytes
after close: 0 live bytes
```

The point is the second line: `lua_close` returned every byte.

3. The virtual stack

Lua and C disagree about types and about memory. *Programming in Lua* says a C type that could hold any Lua value would have two drawbacks, and the second is memory: "If we keep a Lua value in a C variable, the Lua engine has no way to know about this use; it may (wrongly) assume that this value is garbage and collect it." (*PiL 1st ed.*, §24.2) The answer is a stack: "Whenever you want to ask for a value from Lua (such as the value of a global variable), you call Lua, which pushes the required value on the stack. Whenever you want to pass a value to Lua, you first push the value on the stack, and then you call Lua (which will pop the value)." And "because this stack is managed by Lua, the garbage collector knows which values C is using."

Indices. "A positive index represents an absolute stack position, starting at 1 as the bottom of the stack; a negative index represents an offset relative to the top of the stack." So "index 1 represents the first element" while "index -1 also represents the last element (that is, the element at the top)." "LUA_MINSTACK is defined as 20, so that usually

you do not have to worry about stack space unless your code has loops pushing elements onto the stack." Only loops that push need `lua_checkstack`.

```
C
lua_pushnumber(L, 42);           /* index 1 */
lua_pushstring(L, "anne");       /* index 2 */
lua_pushboolean(L, 1);           /* index 3, also -1 */
dump(L, "after 3 pushes");       /* helper: prints each slot's type name */
printf("index -2 = %s\n", lua_tostring(L, -2)); /* same slot as index 2 */
lua_settop(L, 5);                /* growing the top fills new slots with nil */
dump(L, "settop(5)");
lua_pop(L, 3);                   /* macro for lua_settop(L, -4) */
dump(L, "pop 3");
```

```
CODE
after 3 pushes top=3 [number, string, boolean]
index -2 = anne
settop(5)          top=5 [number, string, boolean, nil, nil]
pop 3              top=2 [number, string]
```

The manual agrees: `lua_settop` "sets the stack top to this index. If the new top is greater than the old one, then the new elements are filled with nil," and `lua_pop` "is implemented as a macro over `lua_settop`." One trap:

`lua_tolstring` on a number "changes the actual value in the stack to a string."

4. Giving Lua a C function

The type is `typedef int (*lua_CFunction) (lua_State *L)`; and the protocol is one paragraph: "a C function receives its arguments from Lua in its stack in direct order (the first argument is pushed first). So, when the function starts, `lua_gettop(L)` returns the number of arguments received ... To return values to Lua, a C function just pushes them onto the stack, in direct order ... and returns in C the number of results."

The return value is not a result. It is a count telling Lua how many top slots are results. The stack is private:

"Whenever Lua calls C, the called function gets a new stack, which is independent of previous stacks."

```
C
static int host_average(lua_State *L) {
    int n = lua_gettop(L); /* number of arguments received */
    if (n == 0) return luaL_error(L, "average needs at least one number");
    double sum = 0;
    for (int i = 1; i <= n; i++) {
        sum += luaL_checknumber(L, i); /* raises 'bad argument #i' if not a number */
    }
    lua_pushnumber(L, sum / n); /* first result */
    lua_pushnumber(L, n); /* second result */
    return 2; /* tell Lua: the top two values are results */
}
```

Registering is one call, which the manual defines as `(lua_pushcfunction(L, f), lua_setglobal(L, n))`. After `lua_register(L, "average", host_average)`; the host ran four chunks:

```
CODE
> print(average(10, 20, 60))
30.0 3.0
> print(type(average))
function
> print(average('x'))
error: [string "print(average('x'))"]:1: bad argument #1 to 'average' (number expected, got string)
> print(average())
error: [string "print(average())"]:1: average needs at least one number
```

Both failures are ordinary Lua errors, and `luaL_error` "never returns," hence the idiom `return luaL_error(args)`.

The results print as `30.0` and `3.0` because `lua_pushnumber` pushes a float; `lua_pushinteger` would push an integer.

5. Calling Lua, and why pcall

The calling protocol: "first, the function to be called is pushed onto the stack; then, the arguments to the call are pushed in direct order." On return, "all arguments and the function value are popped and the call results are pushed onto the stack." For failures, the signature is `int lua_pcall (lua_State *L, int nargs, int nresults, int msgch);` and "if there is any error, lua_pcall catches it, pushes a single value on the stack (the error object), and returns an error code."

The first line of the snippet below, `lua_getglobal`, "Pushes onto the stack the value of the global name. Returns the type of that value." The global environment itself lives in the registry, where, at index `LUA_RIDX_GLOBALS`, "the registry has the global environment."

```
C
lua_getglobal(L, "area");           /* function */
lua_pushnumber(L, 3);               /* arg 1 */
lua_pushnumber(L, 4);               /* arg 2 */
int status = lua_pcall(L, 2, 2, 0); /* 2 args in, exactly 2 results out */
printf(" result 1 (index -2) = %g\n", lua_tonumber(L, -2));
printf(" result 2 (index -1) = %g\n", lua_tonumber(L, -1));
lua_pop(L, 2);
```

The full program also called a function that raises an error, and a number that is not a function. Observed output of the whole run:

```
CODE
area status=0 top=2
  result 1 (index -2) = 12
  result 2 (index -1) = 14
boom status=2 (LUA_ERRRUN=2) top=1
  error object: [string "function area(w, h) return w * h, 2 * (w + h)..."]:2: kaboom
call a number status=2: attempt to call a number value
final top=0
```

The `top` values are the evidence: success left the two requested results, failure left one error object.

Protection matters because "internally, Lua uses the C longjmp facility to handle errors," and "if an error happens outside any protected environment, Lua calls a panic function (see `lua_atpanic`) and then calls `abort`, thus exiting the host application." Calling the unprotected `lua_call` on a chunk that raises an error did this on Lua 5.4:

```
CODE
calling with lua_call (unprotected)...
PANIC: unprotected error in call to Lua API ([string "error('unprotected failure')"]:1: unprotected failure)
```

The process was terminated by `abort` (the shell reported status 134) and never printed its last line. *Programming in Lua* gives the rule: "When you are writing the main code in an application, you should not use `lua_call`, because you want to catch any errors. When you are writing functions, however, it is usually a good idea to use `lua_call`" (PiL 1st ed., §27.1). A C function that Lua called is already inside a protected call, which is why Section 4's `luaL_error` was safe.

6. Controlling what a script can touch

The first lever is Section 1's: open only the libraries you want. The second is the environment.

The environment. "any reference to a free name ... `var` is syntactically translated to `_ENV.var`," and "Any table used as the value of `_ENV` is called an environment." `load` takes an `env` argument: "its first upvalue is set to the value of `env`." Here is the Lua side, with a script that can see only `print` and `type`:

```
LUA
local env = { print = print, type = type } -- the only names the script can see
local chunk = load("print('inside', type(os)); return os.time()", "=script", "t", env)
print(pcall(chunk))
--> inside nil
--> false script:1: attempt to index a nil value (global 'os')
```

The chunk's free names are looked up in `env`, so `os` is `nil` inside it and the failure is an ordinary error that `pcall` catches. From C the same idea takes two steps, because the manual says "you have to load the chunk and then change the value of its first upvalue; see `lua_setupvalue`." Here the host gives a script an environment holding only `print` and `type`:

```
C
luaL_loadstring(L, script);          /* the chunk is now on the stack */
lua_newtable(L);                     /* env table */
const char *allowed[] = { "print", "type", NULL };
for (int i = 0; allowed[i]; i++) {
    lua_getglobal(L, allowed[i]);     /* copy ONE trusted function in, by name */
    lua_setfield(L, -2, allowed[i]); /* env[name] = value, pops the value */
}
lua_setupvalue(L, -2, 1);             /* first upvalue of the chunk is _ENV: assign env, pop it */
```

The chunk was loaded first, so it sits below the table, and `lua_setupvalue(L, -2, 1)` assigns the table to the chunk's first upvalue, which is `_ENV`, and pops it. The script is `print('inside', type(io), type(os))` then `return io.open('/etc/hostname')`. Observed, sandboxed and then unrestricted:

```
CODE
inside nil nil
sandboxed error: [string "print('inside', type(io), type(os))..."]:2: attempt to index a nil value
(global 'io')
inside table table
unrestricted: status=0, result type userdata
```

An earlier run allowed only `print`, and the script died on `attempt to call a nil value (global 'type')`: the allow-list working.

What it does not stop. An environment limits which names a script can reach. It does not limit how long the script runs: a chunk loaded with an empty environment can still run `while true do end` forever. The manual describes a count hook, "called after the interpreter executes every count instructions," set with `lua_sethook`. No hook was run, so treat it as a pointer, not a tested recipe.

7. What to take away

Lua is a library and the host owns `main`. A `lua_State` is a thread pointing at an independent interpreter, made by `luaL_newstate` with defaults or by `lua_newstate` with your allocator. Values cross the boundary on a private virtual stack, indexed from the bottom by positive numbers and from the top by negative ones, because both the C code and the garbage collector can see it. A C function reads arguments from index 1 and returns how many results it pushed. Calling Lua means push the function, push the arguments, `lua_pcall`, read, pop. A failed protected call leaves one error object; a failed unprotected call kills the host. Restricting a script means opening fewer libraries and handing it a smaller environment: load with an `env` argument from Lua, or `lua_setupvalue` on the chunk's first upvalue from C.

Next: a review of the key ideas of Chapters 1 to 8, with small programs you can run.

Key Terms

- **Host program:** the embedding application, usually written in C, that creates the Lua state and starts all Lua code by calling into it.

- **lua_State**: an opaque structure that points to a thread and indirectly to the whole state of one Lua interpreter, passed as the first argument of nearly every API function.
- **luaL_newstate**: creates a new state using a default ISO C allocator and sets a warning function and a panic function that print to standard error.
- **lua_newstate**: creates a new state from an allocator function and a user data pointer, with no libraries opened and no panic handler installed.
- **luaL_openlibs**: opens all the standard libraries into a state, which a fresh state does not have.
- **Load versus run**: `luaL_loadstring` and `luaL_loadfile` only compile a chunk onto the stack, and `lua_pcall` is what runs it.
- **Virtual stack**: the per-call array of Lua values through which C and Lua exchange everything, visible to the garbage collector because it is part of the state.
- **Stack index**: a positive index counts from the bottom starting at 1 and a negative index counts from the top so that -1 is the top element.
- **lua_CFunction**: a C function of type `int f(lua_State *L)` that reads its arguments from index 1 upward and returns the number of results it pushed.
- **lua_register**: a macro that pushes a C function and stores it as a global, so Lua code can call it by name.
- **lua_pcall**: calls the function on the stack in protected mode, returns a status code, and leaves a single error object on the stack when the call fails.
- **Panic function**: the handler Lua calls when an error happens outside any protected call, after which the host application exits.
- **Environment control**: restricting a script by opening fewer libraries and giving the chunk a smaller environment, with `load` and an `env` argument from Lua or `lua_setupvalue` from C.

Summary

Lua is a library and the host owns main. A `lua_State` is a thread pointing at an independent interpreter, created by `luaL_newstate` with defaults or `lua_newstate` with your own allocator. Values cross the boundary on a private virtual stack indexed from the bottom by positive numbers and from the top by negative ones. A C function reads its arguments from index 1 and returns how many results it pushed. Calling Lua means push the function, push the arguments, `lua_pcall`, then read and pop; a failed protected call leaves one error object and a failed unprotected one aborts the host. Scripts are restricted by opening fewer libraries and giving them a smaller environment.

Review of Key Concepts

Lecture

Why this comes last

A review that asks you to recite definitions measures memory. The skill this book is for is the other one: opening a Lua file you did not write, or wrote a year ago, and explaining what it does. So this lecture gives you a short method and then uses it on eight small programs, one concept from each of Chapters 1 to 8, stopping at the places where that concept is doing the work.

By the end you should be able to:

- Read an unfamiliar Lua file with a four-question method.
- Name the concept at work in a short excerpt and predict what it does.
- Run a one-line test to confirm a prediction instead of arguing about it.
- Connect the central rule of each of Chapters 1 to 8 to a snippet you can run on Lua 5.4.

1. A four-question method

1. **What does the file return?** Lua files end in a `return` (Chapter 6) or build globals. The returned value is the file's public face.
2. **Where are the tables?** Almost every structure is one (Chapter 4): a record, a set, a module, a class. Decide which hat each one wears.
3. **Where are the metatables and colons?** `setmetatable`, `:` and `__index` mean objects (Chapter 5).
4. **Where can it fail or pause?** `pcall`, `error`, `coroutine` and callbacks mark the control flow that is not top to bottom (Chapters 3 and 7).

Apply the method to the eight checks below, one for each of Chapters 1 to 8. In each check, make the prediction first and then run the snippet.

2. Check for Chapter 1: what counts as true, and what kind of number is this?

Chapter 1 gave two facts that keep paying off. The manual states the first: "Both nil and false make a condition false; they are collectively called false values. Any other value makes a condition true." (Lua 5.4 Reference Manual, §2.1) The second is that the number type has two subtypes, integer and float, which print differently.

Check. Predict what each of these prints before you run it:

```
LUA
print(0 and "0 is true", "" and "empty string is true", nil and 1, false == nil)
print(10 / 2, 10 // 2, math.type(10 / 2), math.type(10 // 2))

CODE
0 is true  empty string is true  nil  false
5.0  5  float  integer
```

Zero and the empty string are true, `nil` and `false` are the only false values, and `false == nil` is itself false because they are different values. `10 / 2` is a float and `10 // 2` is an integer.

3. Check for Chapter 2: who owns this variable?

Read the next excerpt with Chapter 2's scope rules. `total` is declared outside `add` and assigned inside it, so the function's `total` is not a new variable. The manual: "A local variable used by an inner function is called an upvalue (or external local variable, or simply external variable) inside the inner function." (Lua 5.4 Reference Manual, §3.5) The second half of the excerpt shows the other face of the chapter, a misspelled name that Lua quietly treats as a global.

```
LUA
local total = 0
local function add(n)
    total = total + n          -- assigns the OUTER local through an upvalue
end
add(5)
add(7)
print(total)

countr = 1                    -- meant "counter": no error, it just makes a global
print(counter, _G.countr)

CODE
12
nil  1
```

Without the `local total = 0` line, `total` would be a global that starts as `nil`, and the first `total + n` would fail. And `countr = 1` raised no error: a free name is rewritten to `_ENV.countr`, so the typo became a new global and `counter` stayed `nil`.

4. Check for Chapter 3: a function that returns two things

`gsub` is Chapter 3's multiple returns in a real library function. The manual says it "also returns, as its second value, the total number of matches that occurred," and on multiple results: "When a multi-res expression is used in a list of expressions without being the last element, or in a place where the syntax expects a single expression, Lua adjusts the result list of that expression to one element." (Lua 5.4 Reference Manual, §3.4.12)

Check. A local that names one variable drops the count:

```
LUA
print(("a b"):gsub("%s+", " "))
local text = ("a b"):gsub("%s+", " ")    -- one variable: the count is dropped
print(text)
print(select("#", ("a b"):gsub("%s+", " ")))

CODE
a b 1
a b
2
```

The first call is the last item in the argument list of `print`, so both results survive. In `local text = ...` there is one variable, so the count is dropped. That is correct here, and it would be a bug in `return text:gsub(...)`, which hands the count to the caller as well. `select("#", ...)` confirms that the call produced two values.

5. Check for Chapter 4: the length trap

A set is the right tool for membership and a list for order, and a list with a hole is a bug waiting for a different build. The manual's definition: "A table with exactly one border is called a sequence," and "When `t` is not a sequence, `#t` can return any of its borders."

Check. What is `#{10, 20, 30, nil, 50}`? The manual says the table has two borders, 3 and 5, so either answer is legal:

```
LUA
local t = {10, 20, 30, nil, 50}
print(#t)
for i, v in ipairs(t) do io.write(v, " ") end
print()

CODE
5
10 20 30
```

On this Lua 5.4 build `#t` printed 5, while `ipairs` stopped at the hole after 30. Do not build anything on the length of a table with a hole, because the manual says the exact border "depends on details of the internal representation."

6. Check for Chapter 5: where does a method come from?

A method call is Chapter 3's colon sugar plus Chapter 5's `__index`. The manual: "The colon syntax is used to emulate methods, adding an implicit extra parameter `self` to the function." (Lua 5.4 Reference Manual, §3.4.11)

Apply method question 3. `function Point:get()` is a field on the `Point` table that receives the object as `self`, and `Point.__index = Point` sends a miss on an instance to the class. Check it with `rawget`, which ignores metamethods:

```
LUA
local Point = {}
Point.__index = Point
function Point.new(x) return setmetatable({ x = x }, Point) end
function Point:get() return self.x end
local p = Point.new(7)
print(p:get(), rawget(p, "get"), rawget(Point, "get") ~= nil)
```

FA26 | 2026

```
CODE
7 nil true
```

The method is not on the instance (`rawget` gives `nil`) but is on the class.

7. Check for Chapter 6: does require run the file twice?

The manual defines the loading side: "If the loader returns any non-nil value, `require` assigns the returned value to `package.loaded[modname]`," and "The function starts by looking into the `package.loaded` table to determine whether `modname` is already loaded. If it is, then `require` returns the value stored at `package.loaded[modname]`." (Lua 5.4 Reference Manual, `require`) To keep the check self-contained, register the module in `package.preload` instead of in a file:

```
LUA
package.preload["counter"] = function()
  print("counter module runs")
  local M = { n = 0 }
  function M.bump() M.n = M.n + 1; return M.n end
  return M
end
local a = require("counter")
local b = require("counter")
print(a == b, package.loaded["counter"] == a)
print(a.bump(), b.bump())

CODE
counter module runs
true true
1 2
```

The module body ran once, both names point at the same table, and the table keeps its state between calls. That is the caching that lets a module keep settings in one table and edit them live.

8. Check for Chapter 7: a countdown that keeps its place

A coroutine keeps its place between resumes, so a countdown does not need its state stored anywhere else. The manual: "Lua supports coroutines, also called collaborative multithreading ... a coroutine only suspends its execution by explicitly calling a `yield` function." (Lua 5.4 Reference Manual, §2.6) Here is a countdown as a coroutine, with the observed output:

```
LUA
local time_remaining = 3
local function ticker()
  while time_remaining > 0 do
    time_remaining = time_remaining - 1
    coroutine.yield(time_remaining)
  end
  return "done"
end
local co = coroutine.create(ticker)
while true do
  local ok, v = coroutine.resume(co)
  print(ok, v, coroutine.status(co))
  if coroutine.status(co) == "dead" then break end
end

CODE
true 2 suspended
true 1 suspended
true 0 suspended
true done dead
```

Each `resume` returns `true` plus the yielded value. The final line is the body returning, and `status` reports `dead`.

9. Check for Chapter 8: what the host sees

Chapter 8 showed the host's side: a script runs inside whatever environment the host gives it, and an error comes back to a protected caller as an object. Lua can play host to itself with `load` and `pcall`. The manual describes the host's position: "Usually, this call is protected; so, when an otherwise unprotected error occurs during the compilation or execution of a Lua chunk, control returns to the host, which can take appropriate measures, such as printing an error message."

Check. Give a chunk an environment that holds only `print` and `type`, then let it reach for `os`:

```
LUA
local env = { print = print, type = type } -- the only names the script can see
local chunk = load("print('inside', type(os)); return os.time()", "=script", "t", env)
print(pcall(chunk))

CODE
inside nil
false script:1: attempt to index a nil value (global 'os')
```

The chunk's free names resolve in `env`, so `os` is `nil` inside it, the failure comes back as ordinary values from `pcall`, and the calling program carries on. That is Chapter 2's `_ENV` rule and Chapter 8's sandbox in three lines.

10. What to take away

Reading Lua is mostly pattern recognition: what does the file return, which tables are modules, sets or classes, where are the colons and metatables, and where does control leave top to bottom order. Eight checks tied the book together. Only `nil` and `false` are false, and a number is an integer or a float (Chapter 1). A closure assigns the outer local it names, and a misspelled name becomes a global (Chapter 2). A `gsub` returns two values and a local keeps one (Chapter 3). `#` on a list with holes may answer either border (Chapter 4). A colon method is a class field found by `__index` (Chapter 5). `require` caches one table per module (Chapter 6). A coroutine keeps a loop's state on its own stack (Chapter 7). A script sees only the environment its host gives it (Chapter 8). Wherever a prediction was in doubt, a one-line test settled it.

Key Terms

- **Four question method:** ask what the file returns, where the tables are, where the metatables and colons are, and where control can fail or pause.
- **Truthiness test:** if `x` then treats only `nil` and `false` as false, so zero and the empty string count as true.
- **Integer and float subtypes:** the number type holds both, and they print differently: `10 / 2` is the float 5.0, while `10 // 2` is the integer 5.
- **Upvalue assignment:** an inner function assigning to an outer local, which changes that local rather than creating a new variable or a global.
- **Misspelled global:** a name not declared local is a free name, so a typo reads as `nil` or creates a new global without any error.
- **gsub second result:** `string.gsub` returns the substituted string and the match count, and a local declaring one variable drops the count.
- **Border:** a valid length of a table, and a list with a hole has more than one, so the length operator may return any of them.
- **Colon method as class field:** function `Class:name()` is a field named `name` whose first parameter is `self`, found by an instance through the `__index` chain.
- **Module caching:** `require` stores a module's returned value in `package.loaded`, so every later `require` of that name returns the same table without rerunning the file.

- **In-memory module:** a loader function stored in `package.preload`, which require runs like any other loader so the module need not be a file.
- **Coroutine countdown:** the same loop written with `coroutine.yield`, where `resume` returns true plus the yielded value and status reports dead after the body returns.
- **Restricted environment:** a table passed as the `env` argument of `load`, which becomes the chunk's `_ENV` so that names outside it are nil.
- **Protected call:** `pcall`, or `lua_pcall` from C, returns an error to its caller as a value instead of stopping the program.

Summary

Reading Lua is pattern recognition: what the file returns, which tables are modules, sets or classes, where the colons and metatables are, and where control leaves top to bottom order. Eight checks, one for each of Chapters 1 to 8, tied the book together: only nil and false are false and numbers have integer and float subtypes, a closure assigns the outer local it names, `gsub` returns two values, a list with holes may report either border, a colon method is found by `__index`, `require` caches one table per module, a coroutine keeps a loop's state on its own stack, and a script sees only the environment it is given.