Lua in a day

This Course is based on

- Programming in Lua,
  book by Roberto Ierusalimschy
- Lua course by Fabio Mascarenhas
- Beginning Lua Programming,
  book by Kurt Jung, Aaron Brown

Audience

Aimed for audience with at least one of the criteria below:

- Some programming experience
- Some experience with scripting language
- Some experience with language that has functions and data structures as *first-class* values

Agenda

What and why use Lua?
Setup the environment
Lua basics
Control Flow
- Exercise round 1

Functions
not - and - or, useful idioms
- Exercise round 2

Data structures
More about functions
Iterators
- Exercise round 3

Coroutines
- Exercise

Error handling
Metatables
- Exercise

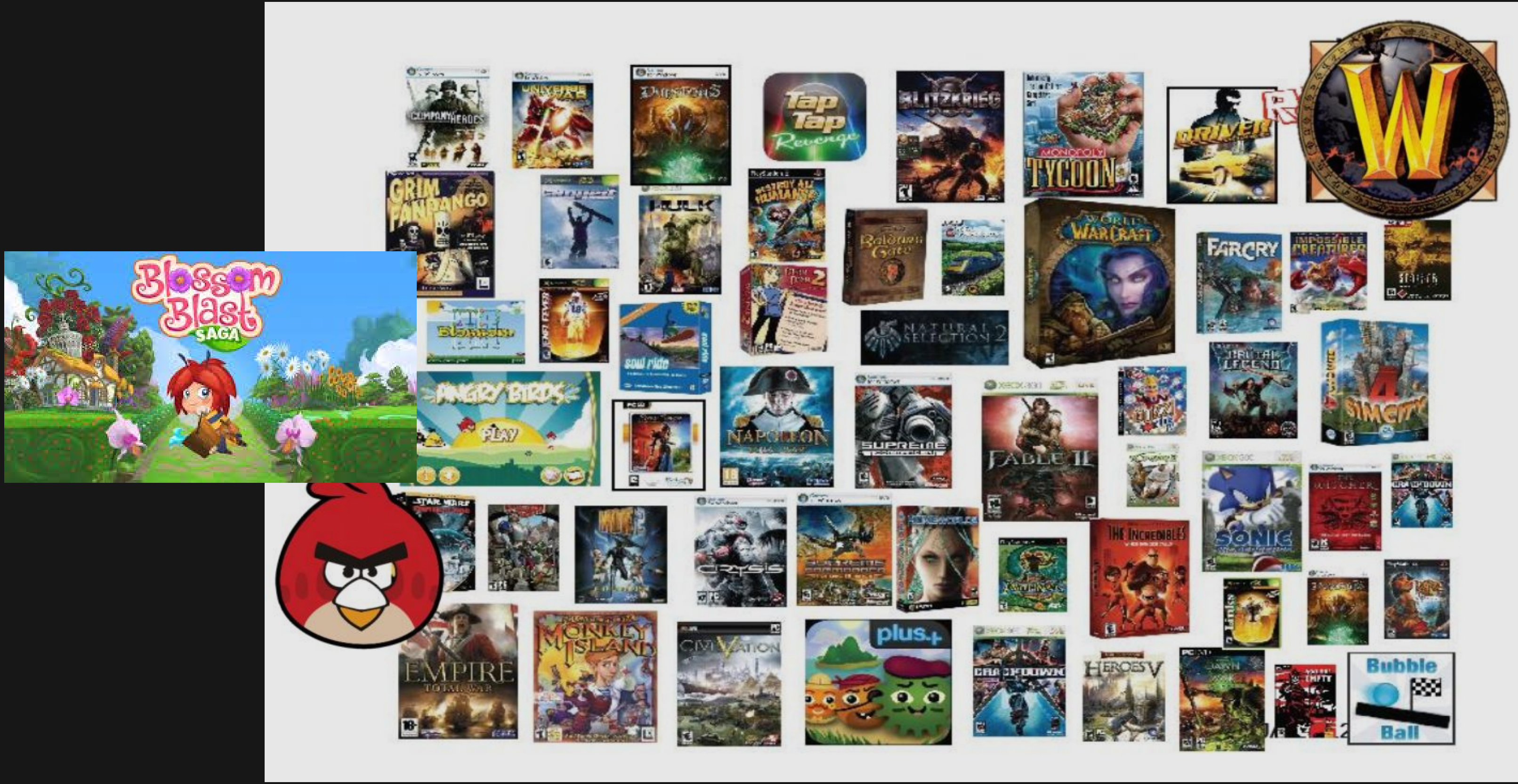
Loading and running code
Standard libraries
- Exercise

Useful tools

What is Lua?

- Scripting language, similar to Python, Ruby and JavaScript
- Lua is widely used in games
- At King, Lua is used:
  - For all code in projects built on top of the Defold engine
  - As extension for parts of Fiction Factory engine
  - Plazma

Games programmed in Lua



Why use Lua?

- Lua is Fast
  - Beats almost every other scripting language
  - Near C performance in many cases
- Lua is Portable
  - Runs on all flavors of Windows, Uni and microprocessors
- Lua is embeddable
  - Easy to extend programs written in other languages
- Lua is small
  - Add between 300kb to 600kb to main executable
- Lua is free
  - Open-source MIT license

Lua versions

- Lua 5.1 (2006)
- LuaJIT 2.0 (2012) -> Lua 5.1 that is:
  - Optimized for each platform
  - Improved in performance
  - Some Lua 5.2 language features (bitop)
  - Used in Defold
- Lua 5.2 (2012) -> has bitwise operations
- Lua 5.3 (2015) -> has new numerical types

Setup your environment for the exercises

You have couple of options

Option A: Use Defold Editor

Option B: Use online terminal / IDE

Lua Online Terminal

Lua Online IDE with project, folder and file structure

It's worth noting that on the website there's also an interactive Lua tutorial

Option C: Setup a Lua dist

- LuaJIT
  or
- Lua 5.1 (5.2 / 5.3) on your computer

Good Text Editors / IDEs for Lua development

Sublime Text with plugins
Atom with plugins
IntelliJ with plugin Sylvanaar
Vim
ZeroBrane Studio
Visual Studio Code
Defold Editor (2 soon)
LDT based on Eclipse
Lua Glider

Sublime

FOLDERS

- example_app
  - .git
  - .internal
  - config
  - examples
    - character_abilities
    - end_game_purchase
    - game_session
    - helpers
    - mercado_store
    - message_center
    - multi_friend_selector
    - omniverse_openworld
    - ota_translations
    - out_of_lives
    - settings_menu
    - slayer_properties
    - start_menu
    - toplist
    - toplist_periodic
    - vanilla_store
      - dksdk.script
      - landing.gui
      - landing.gui_script
      - store_hc.gui
      - store_hc.gui_script
      - store_sc.gui
      - store_sc.gui_script
      - vanilla_store.collection
    - examples.collection
    - examples.gui
    - examples.gui_script
    - examples.lua
  - jenkins
  - locale
  - tests
  - tratt-server
  - .gitignore
  - game.project
  - git.setup
  - ldoc.css
  - ldoc.ld
  - ldoc.ltp
  - log.txt

store_hc.gui_script

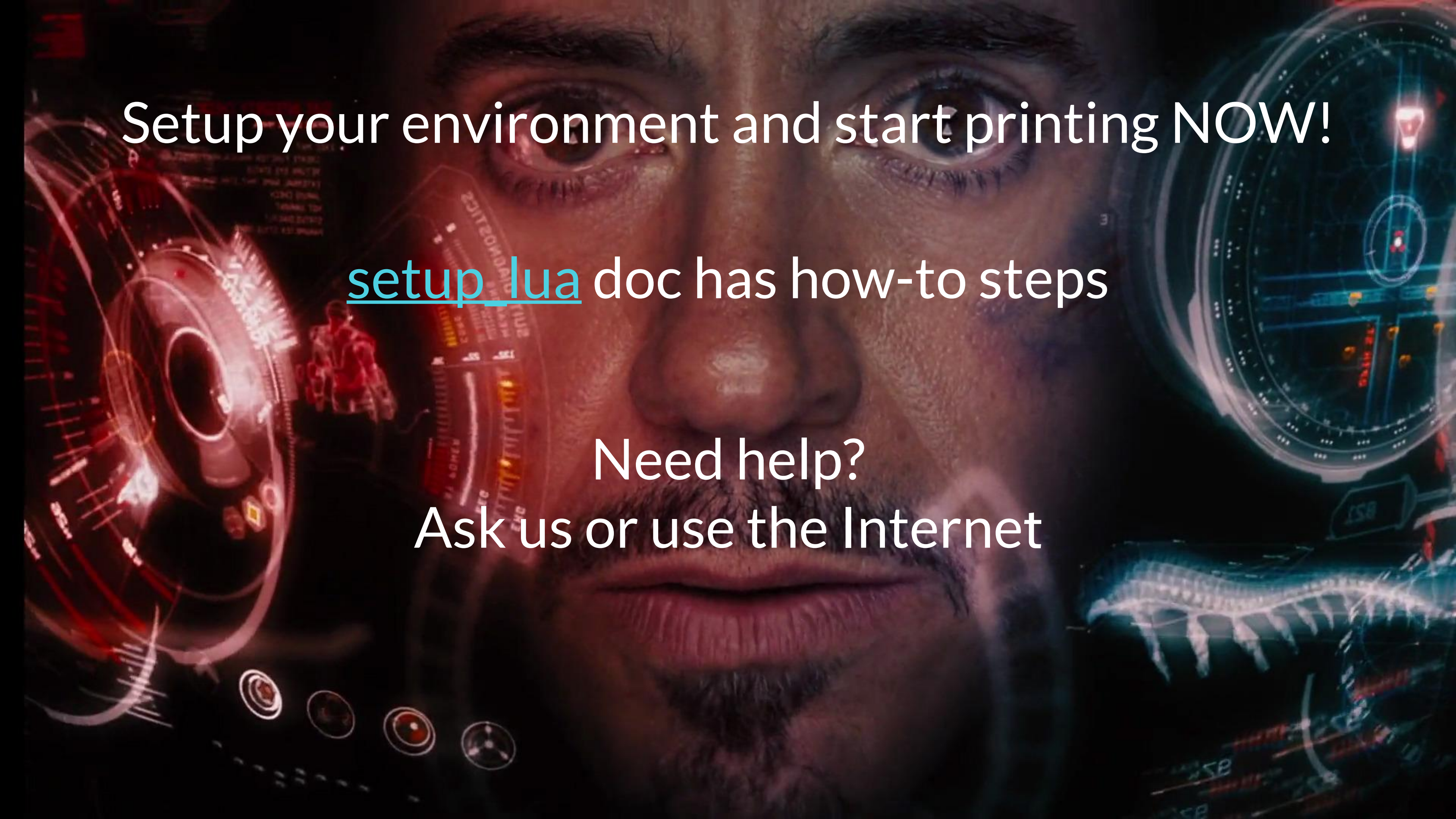
```lua
local gui_types = require "king_constants.gui"
local gui_elements = require "king_constants.gui_element"
local product_package_type = require "king_constants.product_package_type"

local log = logger.create("[VANILLA_STORE_HC]")

local MSG_SHOW = hash("show")

local RESULT_TITLE_SUCCESS = "Your purchase was successful!"
local RESULT_TITLE_FAILURE = "Purchase Failed"
local RESULT_DESCRIPTION_SUCCESS = "You now have more currency"
local RESULT_DESCRIPTION_FAILURE = "For some reason the purchase\n was not completed.\nNo hard currency was added."
local RESULT_TITLE_FAILURE_CONNECTION = "Connection Lost"
local RESULT_TITLE_FAILURE_SETUP = "Store Failure"
local RESULT_DESCRIPTION_FAILURE_CONNECTION = "You are not connected to the internet.\nPlease try again."
local RESULT_DESCRIPTION_FAILURE_SETUP = "Could not communicate with store %s"

a_global_variable = "is a big no no (will be covered in presentation shortly)"

local function is_debug_khc()
    return sys.get_config("dev.debug_khc", 0) == "1"
end

--- Filters products by min_amount
-- @param products
-- @param min_amount
-- @return a new indexed table of filtered products by min_amount
local function get_filtered_products(products, min_amount)
    local filtered_products = {}
    for _, product in pairs(products) do
        if product.iap and product.amount >= min_amount then
            table.insert(filtered_products, product)
        end
    end
    return filtered_products
end

local function sort_products(products)
    table.sort(products, function(a, b)
        return a.amount < b.amount
    end)
end

local function setup_product_nodes(self)
    for _,entry in pairs(self.entries) do
        gui.delete_node(entry.root_node)
        gui.delete_node(entry.button.gui_node)
    end

    self.entries = {}
    self.store_products = store.get_products()
    local products = get_filtered_products(self.store_products, self.min_amount)
    sort_products(products)
```

STATUS: example_app

```
BRANCH:  On branch `develop` tracking `origin/develop`.
ROOT:    ~/code/Defold/branches/4378/1080/example_app
HEAD:    f5e2359 still testing github jenkins hook

UNSTAGED:
  examples/vanilla_store/store_hc.gui_script

STASHES:
  (0) a4a783b testing smu

###################                   ###############
## SELECTED FILE ##                   ## ALL FILES ##
###################                   ###############

[o] open file                         [a] stage all unstaged files
[s] stage file                        [A] stage all unstaged and untracked files
[u] unstage file                      [U] unstage all staged files
[d] discard changes to file           [D] discard all unstaged changes
[h] open file on remote
[M] launch external merge tool for conflict

[l] diff file inline                  [f] diff all files
[e] diff file                         [F] diff all cached files

#############                         #############
## ACTIONS ##                         ## STASHES ##
#############                         #############

[c] commit                            [t][a] apply stash
[C] commit, including unstaged        [t][p] pop stash
[m] amend previous commit             [t][s] show stash
[P] push current branch               [t][c] create stash
                                      [t][u] create stash including untracked file
[i] ignore file                       [t][d] discard stash
[I] ignore pattern

###########
## OTHER ##
###########

[r]           refresh status
[?]           toggle this help menu
[tab]         transition to next dashboard
[SHIFT-tab]   transition to previous dashboard
[.]           move cursor to next file
[,]           move cursor to previous file

-
```

Comparing against : HEAD, develop*, Line 50, Column 1

Tab Size: 2   GitSavvy Status



Setup your environment and start printing NOW!

setup_lua doc has how-to steps

Need help?
Ask us or use the Internet

Lua basics

Chunks

Each piece of code that Lua executes is a *chunk*

A chunk is simply a sequence of statements

Statements

A Lua program is composed of *statements*:

- assignments
- conditionals
- loops
- function calls
- ...

Separating Statements

The syntax does not need separators
to know when a statement ends,
even if they are on the same line:

```
a = 1 b = 2 print(a, b)  -->   1     2
```

Identifiers

A name of a variable, function, or other user defined item

`tony`	`tony_stark`	`tonyStark`
`_tony`	`IronMan9`	`_`

Starts with a letter or '_' followed by 0 or more letters, underscores, or digits

Keywords

Can not be used as identifiers

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

Comments

```
-- a comment that runs until end of line
single = 1


--[[ a multi-line comment,
that ends with ]]
multi = 2
```

Global variables

- Variables are *global* by default
- They are available everywhere
- They are not declared, just assigned

Avoid global variables, keep the Verse shiny

Local variables

Prefer creating *local* variables instead of global ones

```
local function greet(name)
    return "Good morning " .. name
end

local the_greet = greet("Zoe")
print(the_greet)  -->   Good morning Zoe
```

Types and values

No type definitions, each value carries it's own type

Types: *userdata*, *thread*, and the 6 below

```
type(true)     -->  boolean
type(nil)      -->  nil
type(2.3)      -->  number
type(type)     -->  function
type("shiny")  -->  string
type({key = "value"})  --> table
```

Booleans

Any non-boolean value evaluates to *true* except for *nil*

This gives 2 cases that are different from other languages:

```
if 0 then
    print("this line will execute")
end

if "" then
    print("this line will also execute")
end
```

Logical operators

Lua	**C++**
`and`	`&&`
`or`	`||`
`not`	`!`

Numbers

- *number* is a double precision floating point
- Represents 32-bit integer without rounding issues

```
-- edge cases
print(3.6 / 0)   --> inf
print(0 / 0)  --> nan
```

Arithmetic and relational operators

==   ~=   <   <=   >=   >

+   -   *   /   ^   %

Strings

string is an immutable sequence of bytes

Example of built-in operations:

```
print("River " .. "Tam")  --> River Tam
print(#"Zoe")    --> 3
```

The string library provides other operations

Conversions

Lua provides automatic conversions
between numbers and strings

```
print("10" .. 1)      --> 101
print("4" * "2")      --> 8

print("Zoe" + 1) --> error, can't do math op. on a str
print("Zoe" .. true) --> error, can't concat. a bool

-- to convert explicitly, use tostring or tonumber
print("Zoe" .. tostring(true))     --> Zoetrue
```

Tables

- Tables are the only way to structure data in Lua
- Tables are associative arrays,
  a collection of key, value pairs
- Any value except *nil* can be a key
- Tables can represent arrays, records, sets, objects (in the OO sense), modules and more

Table Constructors

```lua
-- table with keys as numbers (can be called an array)
local array = {"Zoe", "Tam"}

-- same as
array = {[1] = "Zoe", [2] = "Tam"}
print(array[2])        --> "Tam"

array[3] = "Mal"  -- Add index 3
array[2] = nil         -- remove value in index 2
print(array[2])        --> nil
```

Table Constructors

```lua
-- table with keys as strings (can be called a record)
local record = {my_key = 2}

-- same as
record = {}
record.my_key = 2

-- and same as
record = {}
record["my_key"] = 2
```

Table Constructors

```
-- keys can also be of other types
-- keys can also be variables

local array = {"Zoe", "Tam"}
local record = {my_key = 2}

local t = { [record] = "key is my record table",
            [array] = "key is my array table" }

print(t[record])     --> key is my record table
```

Tables and functions will be discussed more later


Control Flow

if (then) elseif (then) else

```
local function calculate(op, a, b)
    if op == "+" then
        return a + b
    elseif op == "-" then
        return a - b
    else
        error("invalid operation")
    end
end
print(calculate("-", 3, 2))   --> 1
```

while

```lua
local i = 1
local sum = 0

while i <= 3 do
    sum = sum + i
    i = i + 1
end

print(sum)    --> 6
```

repeat until

```
local i = 1
local sum = 0

repeat
    sum = sum + i
    i = i + 1
until i > 3

print(sum)  --> 6
```

Numeric for

```
for i = 1, 3 do
    print(i)     --> 1    2    3
end
```

Numeric for

```
for i = 1, 6, 2 do
    print(i)     --> 1    3    5
end
```

Numeric for

```
for i = 3, 1, -1 do
    print(i)     --> 3    2    1
end
```

Multiple assignment

```
local a, b = 10, 20
print(a, b)   --> 10  20
```

Can be used to swap values

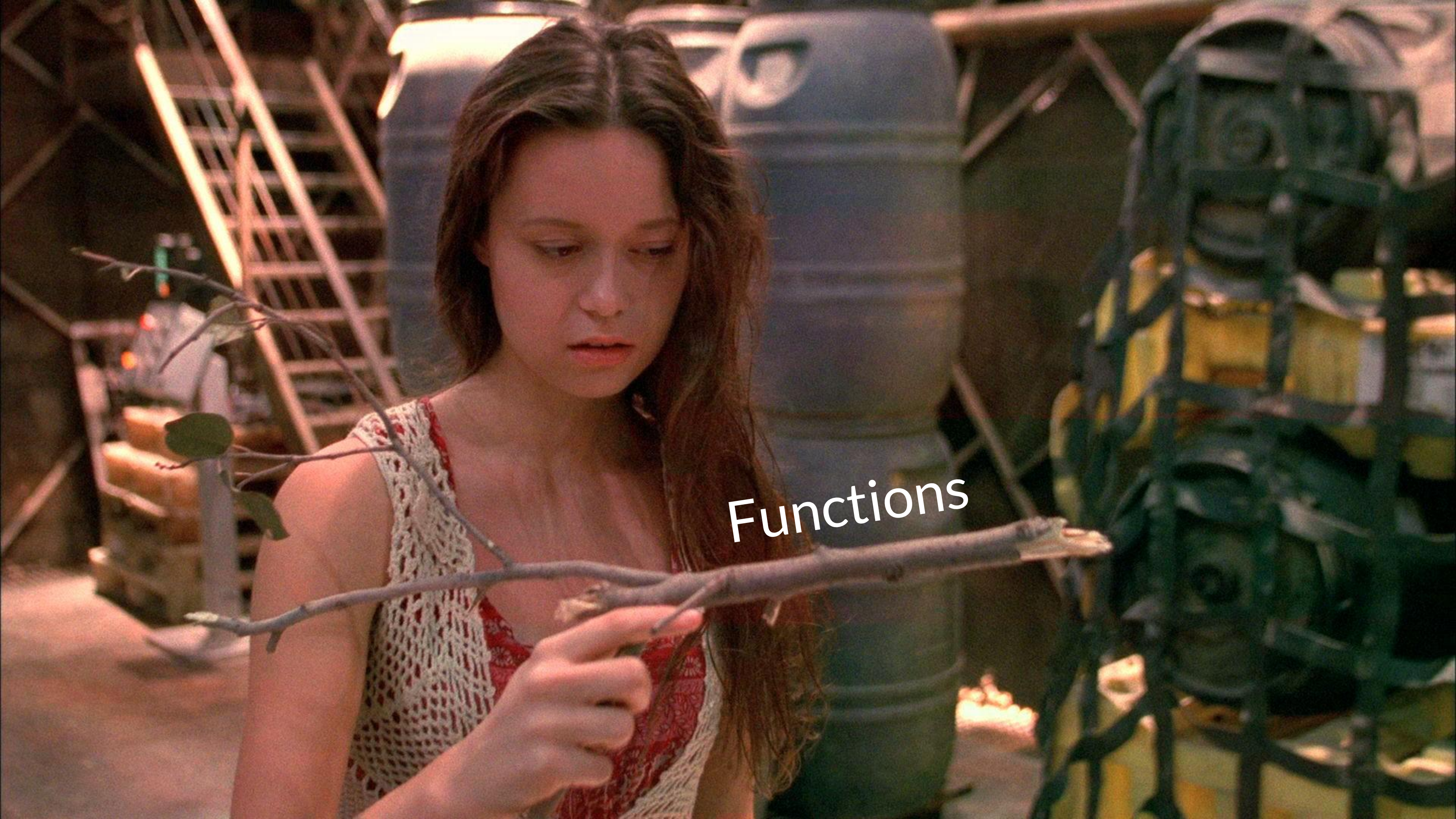
```
local a, b = 10, 20
a, b = b, a
print(a, b)   --> 20  10
```

Multiple assignment

Cases of variables / values mismatch

```
local a, b, c = 10, 20
print(a, b, c)   --> 10  20  nil


local a, b = 10, 20, 30
print(a, b)   --> 10  20
```

Exercise round 1


Functions

Functions

A function abstracts and parameterizes a sequence of statements and expressions

A function call can be a statement or an expression

```
print(8*9, 9/8)                  -- statement
local x = math.sin(3) + math.cos(2) -- expression
print(x, os.date())              -- expression
```

Functions are values

Functions are values, and have no exclusive namespace

```
print(print)   --> function: builtin#29
print = "function no more"
print("this won't work")
--> attempt to call global 'print' (a string value)
```

Be careful to not overwrite / shadow built-in functions

Functions

Local functions are local to the chunk,
just like local variables

The body of a function is a chunk, and the parameters are local variables in this chunk

A *return* statement returns from a function

Named vs Anonymous

```
local function named()
    return function()
            return "this is an anon. func within named func"
        end
end
print(named()()) -->this is an anon. func within named func

-- anonymous function assigned to a variable (expression)
local anon = function() return "anonymous function" end
print(anon())    --> anonymous function
```

About scope

```lua
do_stuff()  -- what happens here ?
local function do_stuff()
   return "do stuff"
end
-- attempt to call global 'do_stuff' (a nil value)

-- under the hood
do_stuff()
local do_stuff = function()
   return "do stuff"
end
```

Using multiple results

```
local function maxmin(a, b)
   if a > b then
      return a, b
   else
      return b, a
   end
end
print(maxmin(2, 3))   --> 3   2
print(maxmin(3, 2))   --> 3   2
```

Using multiple results

When a function call **is last** in a list of expressions,
all of the results of the function will append to the list

```
local a, b, c = maxmin(2, 3)
print(a, b, c)        --> 3  2  nil
a, b, c = 1, maxmin(2, 3)
print(a, b, c)        --> 1  3  2
local function f(x) return maxmin(x, 0) end
print(f(-4))          --> 0  -4
```

Using multiple results

When a function call **is not last** in the list,
only the first result of the function will return to the list

```
local a, b = maxmin(2, 3), 5
print(a, b)     --> 3  5
print(maxmin(2, 3), 5)     --> 3  5
print(maxmin(2, 3) + 2)     --> 5
```

Variadic functions

Last parameter of a function can be the token ...

```
local function printf(fmt, ...)
    io.write(string.format(fmt, ...))
end
printf("%s(%d, %d)\n", "maxmin", 2, 3)
--> maxmin(2, 3)
```

Iterating over "..."

You can collect and then iterate over the extra arguments using ... inside a table constructor, like so {...}

```
function add(...)
    local sum = 0
    for _, n in ipairs({...}) do
        sum = sum + n
    end
    return sum
end
```


and
or
not
Logical operators

Logical operators

Lua	C++
`and`	`&&`
`or`	`||`
`not`	`!`

not

Expression with ***not*** evaluates to *true* or *false*

```
local val
if not val then
    print("Execution will flow here")
end
```

and

and gives its 1st argument if it is *false*

otherwise it gives its 2nd argument

```
print(nil and 2)  --> nil
print(2 and nil)  --> nil
print(9 and 2)    --> 2
print(2 and 9)    --> 9
```

or

or gives its 1st argument if it is *true*

otherwise it gives its 2nd argument

```
print(nil or 2)   --> 2
print(2 or nil)   --> 2
print(9 or 2)     --> 9
print(2 or 9)     --> 2
```

Default parameter

Use ***or*** operator for default parameter functionality

```
local function greet(word)
    local word_used = word or "Shiny"
    print(word_used .. " Verse!")
end
greet()          --> Shiny Verse!
greet("Hello") --> Hello Verse!
```

Ternary operator

Use ***and*** with ***or*** for ternary operator functionality

```
local function max(a, b)
    return (a > b) and a or b
end

max(4, 3)
--> Between (a > b) and a, a is returned.
--  Between a or b, a is returned that is 4.

max(3, 4)
--> Between (a > b) and a, (a > b) is returned.
--  Between (a > b) or b, b is returned that is 4.
```

Exercise round 2

Data structures

Data structures

- Tables are the only way to structure data in Lua
- Tables are associative arrays,
  a collection of key, value pairs
- Any value except *nil* can be a key
- Tables are a *mutable reference type*:

```
alias = tab
alias["x"] = "change"
print(tab["x"]) --> change
```

Arrays

A Lua array is a table with values (no *nil* holes) assigned to sequential integer keys, starting with 1 (one)

```
local a = {}
for i = 1, 3 do
    a[i] = math.random(10)
end
-- an array like the previous one
local a = {math.random(10), math.random(10),
        math.random(10)}
```

Length

```lua
local t = {"what's", "my", "length"}
-- #t gives the number of elements of t
for i = 1, #t do
    print(t[i]) --> what's  my  length
end

-- remove the last element
t[#t] = nil
-- add a new element to the end
t[#t + 1] = "name"
-- looping again would print: what's  my  name
```

Table Concat

```
t = {"one, "two", "three"}
print(table.concat(t, ", "))
--> one, two, three
```

Concatenates an array of strings
using an optional separator
(second parameter's default value is "")

Inserting, removing

```
t = {9, 8, 7}
t[#t + 1] = 6
print(table.concat(t, ", ")) --> 9, 8, 7, 6
t[#t] = nil
print(table.concat(t, ", ")) --> 9, 8, 7
t[1] = 10
print(table.concat(t, ", ")) --> 10, 8, 7
t[2] = nil
print(table.concat(t, ", "))
--> 10
```

Lua array should not have *nil* holes

```
with_holes = {10, nil, 7}
print(table.concat(with_holes, ", ")) --> 10
print(#with_holes) --> 1
```

Length stops "counting" when it reaches *nil* value in table
No *nil* holes is part of the Lua array "contract"

table.concat(array), ipairs(array) and other functions expect the Lua array "contract"

table.insert, table.remove

```lua
t = {9, 8, 7}
table.insert(t, #t, 6)
print(table.concat(t, ", ")) --> 9, 8, 7, 6
table.remove(t, #t)
print(table.concat(t, ", ")) --> 9, 8, 7
table.insert(t, 1, 10)
print(table.concat(t, ", ")) --> 10, 9, 8, 7
table.remove(t, 2)
print(table.concat(t, ", ")) --> 10, 8, 7
```

Iteration with ipairs

```
local a = {1, 3, 5, 7, 9}
local sum = 0
for i, v in ipairs(a) do
    print("index: " .. i ..", value: " .. v)
    sum = sum + v
end
print("the sum is " .. sum)
```

Records

A Lua record is a table with string keys, where the keys are valid identifiers

```lua
p1 = {x = 10, y = 20}
p2 = {x = 50, y = 5}
line = {from = p1, to = p2, color = "blue"}
line.color = "red"
-- same as line["color"] = "red"
print(line.from.x, line["color"])
```

Sets

Sets in Lua is a table where the keys are the elements of the set, and the values are *true*

```
a_set = {[math.random(100)] = true,
         [math.random(100)] = true}
```

If we replace *true* by a number and use it as a counter we have a *multiset*

Iteration with pairs

```
local tab = { x = 5, y = 3, 10, "foo" }
for k, v in pairs(tab) do
    print(tostring(k) .. " = " .. tostring(v))
end
--> 1 = 10    2 = foo    x = 5    y = 3
```

pairs does not guarantee an order of iteration, even among numeric keys

--> use *ipairs* when iterating over an array


More about functions

Lexical scoping

Any local variable visible in the point
where a function is defined, is also visible inside the function

```
local function derivative(f, dx)
    dx = dx or 1e-4
    return function(x)
            -- both f and dx visible here!
            return (f(x + dx) - f(x)) / dx
        end
end
local df = derivative(function(x) return x * x * x end)
print(df(5))    --> 75.001500009932
```

Closures

```
function counter()
    local n = 0
    return function()
        n = n + 1
        return n
    end
end
```

```
a = counter()
b = counter()
print(a()) --> 1
print(a()) --> 2
print(b()) --> 1
```

Each call to counter() creates a new closure
Each closure closes over a different instance of n

Closures can share

```
function counter()
    local n = 0
    return function(x)
        n = n + x
        return n
    end,
    function(x)
        n = n - x
        return n
    end
end
```

```
inc, dec = counter()
print(inc(5)) --> 5
print(dec(2)) --> 3
```

The only way to access *n* is through the closures

Callbacks

Lua closures are a lightweight mechanism for callbacks

```
fighters = {"Mal", "Zoe", "Tam"}
table.sort(fighters, function(a, b)
        return a < b
    end)
print(table.concat(fighters, ", "))
--> Mal, Tam, Zoe
```

Functional programming

- Functional programming (FP) is based on a premise:
  constructing programs using functions
  that have no *side effects*
- Lua is in essence an *imperative* language, so FP is not the usual method, but it can be applied using Lua
- However we will not go into FP in this course


Iterators

Generic for

We have seen how to use the *generic* for loop (or the for-in loop) using the *ipairs* and *pairs* function

The Lua standard library defines other functions that work with generic for:

```
-- for each line in "foo.txt" do...
for line in io.lines("foo.txt") do
    -- for each word in line do...
    for word in string.gmatch(line, "%w+") do
        print(word)
    end
end
```

All these functions have one thing in common: they return *iterators*

Iterators

- An iterator is a function that, each time it is called, produces one or more values that correspond to an item from some sequence
  - Each index and value of an array table
  - Each key and value from a record table
  - Each line from a file
  - Each substring that matches a pattern

- When there are no more items the iterator returns *nil*

Generic for and iterators

The *generic for* will repeatedly call the *iterator* function, assigning the values it returns to the control variables, until the *iterator* returns *nil*

```
local function from_one_to_ten()
    local x = math.random(4)
    if x == 4 then
        return nil
    else
        return x
    end
end
for n in from_one_to_ten do
    print(n)  --> 1  3  1
end
```

Closure iterators

The simplest way to define an useful iterator is to use a closure:

```
function fromto(a, b)  -- this is a stateful iterator
    return function ()  -- where a, b are the state
        if a > b then
            return nil
        else
            a = a + 1
            return a - 1
        end
    end
end
for i in fromto(2, 5) do
    print(i)  --> 2  3  4  5
end
```

Exercise round 3

coroutines

ALERT STATUS
MIKE LEVEL
WHITE
YELLOW
BLUE
RED
SECTION STATUS
ON DECK
SELECTED
MISSILE REJECT
FIRE
LAUNCH ORDER
MISSILE AWAY
NOT READY
LAUNCHER READY
MISSILE READY TO FIRE
LAUNCHER DESIG
SECTION READY
1

Collaborative multi tasking

Coroutines allow you to run a line of execution with it's own stack, local variables and instruction pointer

As opposed to threads only a single coroutine can be run at a time

A coroutine will be run until it explicitly requests to be suspended

Creating a coroutine

Create a coroutine using `coroutine.create(fn)`, passing a function that the coroutine will run:

```
local co = coroutine.create(function()
   print("Hi")
end)
print(co)     --> thread: 0x8071d98
```

Coroutine states

A coroutine can be in one of three states*:

1. suspended
2. running
3. dead

When a coroutine is created it starts in the suspended state

This means it will not run its body automatically when created

* There is a fourth state “normal” but we can ignore that for now

Check coroutine status

Use `coroutine.status(co)` to check the state of a coroutine:

```
print(coroutine.status(co))    --> suspended
```

Start or resume a coroutine

Use `coroutine.resume(co)` to (re)start the execution of a coroutine:

```
local co = coroutine.create(function()
    print("Hi")
end)
print(coroutine.status(co))      --> suspended
coroutine.resume(co)             --> Hi
print(coroutine.status(co))      --> dead
```

Suspend a running coroutine

The real power of coroutines comes from the `yield()` function which suspends execution until resumed:

```
local co = coroutine.create(function()
    for i=1,10 do
        print("co", i)
        coroutine.yield()
    end
end)
coroutine.resume(co)           --> co     1
print(coroutine.status(co))  --> suspended
```

Suspend a running coroutine

When we resume the coroutine the call to `yield()` returns and execution is continued until next `yield()` or program end:

```
coroutine.resume(co)           --> co     2
coroutine.resume(co)           --> co     3
...
coroutine.resume(co)           --> co     10
coroutine.resume(co)           --> prints nothing
print(coroutine.status())      --> dead
print(coroutine.resume(co))    --> false  cannot resume dead
                                          coroutine
```

Protected mode

Note that `resume()` runs in protected mode

If there is an error in a coroutine Lua will not show the error message

It will instead return it to the resume call

```
print(coroutine.resume(co))  --> false  cannot resume dead
                                         coroutine
```

Passing data to/from coroutines

The first `resume()` has no matching `yield()` waiting for it

Additional arguments to `resume()` will pass these to the coroutine main function:

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

Passing data to/from coroutines

A call to `resume()` returns, after the `true` that signals no errors, any arguments passed to the corresponding `yield()`:

```
local co = coroutine.create(function(a, b)
    coroutine.yield(a + b, a - b)
end)
print(coroutine.resume(co, 20, 10)) --> true   30  10
```

Passing data to/from coroutines

Symmetrically a call to `yield()` returns any extra arguments passed to the corresponding `resume()`:

```
local co = coroutine.create(function(a, b)
    print("co", a, b)
    a, b = coroutine.yield()
    print("co", a, b)
end)
coroutine.resume(co, 1, 2)   --> co 1   2
coroutine.resume(co, 3, 4)   --> co 3   4
```

Coroutines as iterators

Coroutines can also be used as iterators

coroutine.wrap(fn) will create a coroutine and return a function that when called will resume the coroutine

Read more about this in chapter 9.3 of Programming in Lua

Exercise - coroutines


Error handling

Error handling

In many applications there is no need to do any error handling in Lua

If there is an error, the application embedding Lua will receive an error code when Lua is asked to run a chunk

If you need to handle errors in Lua use `pcall()`

If you need to raise an error in Lua use `error()`

Example

```
function foo()
  if unexpected_condition then error() end
  print(a[i]) -- potential error: ‘a’ may not be a table
end

if pcall(foo) then
  -- no error
else
  -- foo raised an error
end
```

pcall(f, arg1, ...)

`pcall(f, arg1, ...)` calls its first argument `f` in protected mode

Additional arguments to `pcall()` are forwarded

If there are no errors `pcall()` returns `true` plus any values returned by the call

If there are errors `pcall()` returns `false` plus an error message

error(message)

`error(message)` terminates the last protected function and returns its first argument as error message

```
local ok, error_message = pcall(function()
  error({ msg = "foobar", code = 123})
end)

print(ok, error_message.msg, error_message.code)
--> false    "foobar"    123
```

Note that the error message doesn't have to be a string!

error(message [, level])

`error()` will provide information about the location of the error

```
a = "a" + 1
--> stdin:1: attempt to perform arithmetic on a string value
```

The location gives the filename and line number

error(message [, level])

Use `level` to help Lua “point the finger” in the right direction in the call hierarchy:

```
function test_string(s)
  -- the error should point to the caller of the function
  if type(s) ~= "string" then error("Not a string", 2) end
end
```

assert(v [, message])

Issues an error when its first argument is `false` (ie `nil` or `false`).

The specified message will be passed to the error:

```
assert(1 + 1 == 3, "Impossible math")
--> stdin:1: Impossible math
```

assert(v [, message])

If the first argument is `true` then assert returns all its arguments

A typical idiom to check for errors is:

```
local f = assert(io.open(filename, mode))
```

If `io.open` fails, the error message goes as the second argument to `assert`, which then shows the message

debug.traceback()

Use `debug.traceback()` to generate a traceback of the current execution:

```
local function a()
    print(debug.traceback())
end
a()

stack traceback:
    test.lua:2: in function 'a'
    test.lua:9: in main chunk
    [C]: in ?
```

xpcall(f, err)

Similar to `pcall()`

`f` will be called in protected mode

If an error occurs the `err` function will be called with the error object

Metatables

The predictability of tables

Lua tables usually have a predictable set of operations

- Add key-value pairs
- Check the value associated with a key
- Traverse all key-value pairs

We cannot add, compare or call tables

Metatables

Metatables allow us to change the behavior of a table, for instance what happens if you try to add two tables

Metatables

When Lua tries to add two tables it checks:

- If either of them has a metatable
- If the metatable contains an __add field
  - __add is what is called a metamethod
  - __add should be a function
- If Lua finds this field it calls the corresponding value to compute the sum

Creating a metatable

Lua creates tables without metatables:

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

Use `setmetatable()` to set or change the metatable:

```
local mt = { __add = function(t1, t1) end }
local t = {}
setmetatable(t, mt)
print(getmetatable(t) == mt) --> true
```

Arithmetic metamethods

Each arithmetic operator has a corresponding field name in metatables:

- `__add` for addition
- `__sub` for subtraction
- `__mul` for multiplication
- `__div` for division
- `__unm` for negation
- `__pow` for exponentiation
- `__concat` for string concatenation

Relational metamethods

For each relational operator there is a corresponding field name in metatables:

- `__eq` for *equality* (==)
- `__lt` for *less than* (<)
- `__le` for *less or equal* (<=)

Relational metamethods

There are no separate metamethods for the other three relational operators:

- `a ~= b` will be translated to: `not (a == b)`
- `a > b` will be translated to: `b < a`
- `a >= b` will be translated to: `b <= a`

Example: Adding two tables

```
local mt = { __add = function(a, b)
    local res = {}
    for _,v in pairs(a) do table.insert(res, v) end
    for _,v in pairs(b) do table.insert(res, v) end
    return res
end }
local t1 = setmetatable({ "a", "b", "c", "d" }, mt)
local t2 = setmetatable({ "e", "f", "g", "h" }, mt)
local t3 = t1 + t2
print(table.concat(t3, ""))          --> abcdefgh
```

Example: Comparing two tables

```lua
local mt = { __eq = function(a, b)
    if #a ~= #b then return false end
    for i=1,#a do
        if a[i] ~= b[i] then return false end
    end
    return true
end }
local t1 = setmetatable({ "a", "b", "c", "d" }, mt)
local t2 = setmetatable({ "a", "b", "c", "d" }, mt)
print(t1 == t2)           --> true
```

Library defined metamethods

Arithmetic and relational metamethods are for the Lua core

More metamethods exist to extend the functionality of libraries

Table to string

Change the behavior when printing a table using the `__tostring` metamethod:

```
local t = { "a", "b", "c", "d" }
local mt = { __tostring = function(t)
    return table.concat(t, "")
end }
setmetatable(t, mt)
print(t)                     --> abcd
```

Table access metamethods

With the `__index` and `__newindex` metamethods it is possible to completely change how tables are accessed

__index

When trying to access an absent field in a table, the result is nil

Such access triggers Lua to look for an `__index` metamethod to provide the result

`__index` can be a function or a table

Use `rawget(t, i)` to bypass metamethod lookup on table access

__newindex

When trying to assign a value to an absent field in a table Lua will look for a `__newindex` metamethod

If no such method exists the value will be set as usual

If a `__newindex` metamethod exists it will be called instead

Use `rawset(t, k, v)` to bypass metamethod lookup

OOP in Lua

There is no concept of classes and inheritance in Lua

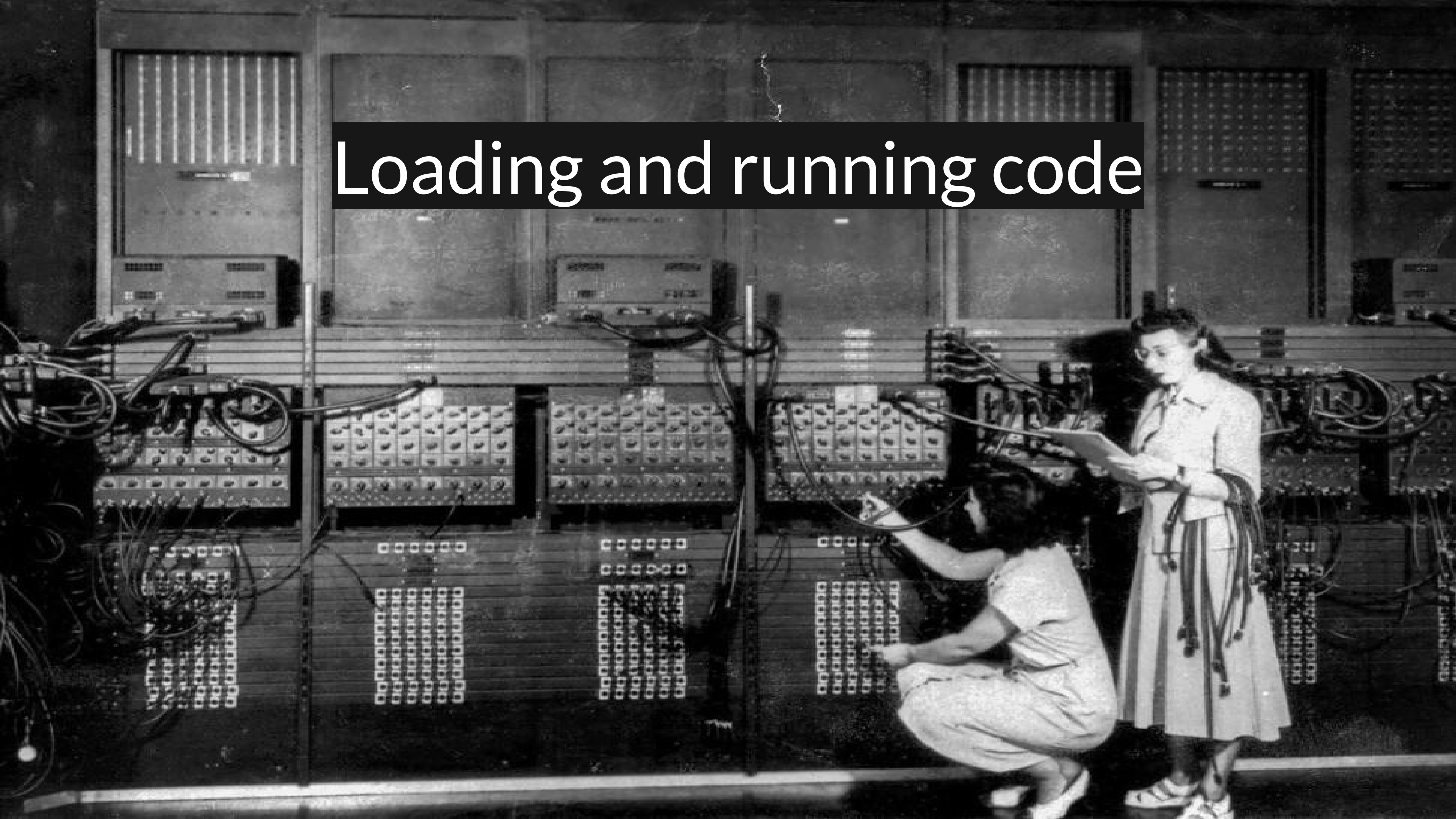
Meta-tables are the foundation on which OOP concepts can be created in Lua

This course will not cover OOP. Further reading:

- Lua wiki on OOP
- 30 Lines of Goodness
- Class Commons

Exercise - metatables and error handling

Loading and running code



Loading code using require()

A Lua module can be used to share code and data between projects or Lua files

A module is loaded using the `require(modname)` function

Loading code using require()

`require()` starts by looking into the `package.loaded` table to see if the module is already loaded

If `package.loaded` already contains a value for `modname` that value will be returned

Otherwise it will try to find a loader for the module and assign the returned value to `package.loaded` for `modname`

Creating a module

The typical way of creating and using a module looks like this:

foo.lua:

```lua
local M = {}

function M.bar()
    print("Foobar")
end

return M
```

something.lua:

```lua
local foo = require "foo"
foo.bar()          --> Foobar
```

Unloading a module

You can unload a previously required module simply by setting its value in `package.loaded` to nil:

```
local foo1 = require("foo")
local foo2 = require("foo")
print(foo1 == foo2)                --> true
package.loaded["foo"] = nil
local foo3 = require("foo")
print(foo1 == foo3)                --> false
```

Loading and running code using dofile()

`dofile(filename)` will open the named file and execute its contents as a Lua chunk

`dofile()` will return all values returned by the chunk

`dofile()` will propagate any errors to its caller

foo.lua:

```
print("foo")
return "bar"
```

test.lua:

```
local res = dofile("foo.lua")--> foo
print(res)                   --> bar
```

Loading code using loadfile()

`loadfile(filename)` will load the named file and return a function that when invoked will run the loaded chunk

All values returned from the chunk will be returned when invoking the function

foo.lua:

```
print("foo")
return "bar"
```

test.lua:

```
local fn = loadfile("foo.lua")
local res = fn()                    --> foo
print(res)                          --> bar
```

Loading code using loadstring()

`loadstring(string)` is similar to loadfile but gets the chunk from the given string

Standard libraries

The standard libraries

Lua has a fairly small standard library

- string
- table
- math
- io
- os
- coroutine
- debug

string.*

Search and replace in strings

`string.find(), string.match(), string.gsub()`

Format

`string.format(), string.lower(), string.upper(), string.reverse(), string.rep()`

Substrings

`string.sub(),string.char()`

table.*

Manipulate Lua tables

```
table.insert(), table.remove()

table.sort()            -- sort either by comparison or using a
                        -- custom search function

table.concat()          -- concatenate into a string
```

math.*

```
math.abs(), math.floor(), math.ceil(),
math.max(), math.min()

math.cos(), math.sin(), math.tan(), ...

math.exp(), math.pow(), math.sqrt()

math.random()
```

io.*

File operations

```
io.open(), io.close(), io.read(), io.write()
```

```
io.lines()    -- read line by line from open file
```

Note that Lua doesn't have any filesystem operations such as getting the files in a directory

Use LuaFileSystem (LFS) to complement existing io functions

os.*

```
os.date(), os.time(), os.clock()

os.exit()

os.getenv()

os.rename(), os.remove()

os.execute()
```

debug.*

Manipulate the Lua environment

```
debug.getinfo()        -- get information about a function (name, line number etc)

debug.traceback()      -- get a traceback of the call stack

debug.getfenv(), debug.setfenv()

debug.sethook()
```

Standard functions

```
assert(), error(), pcall(), xpcall()

collectgarbage()

require(), dofile(), loadfile(), loadstring()

ipairs(), pairs(), next(), select(), unpack()

print()

tostring(), tonumber(), type()

getmetatable(), setmetatable(), rawget(), rawset()
```

<u>Exercise - standard libraries</u>

Useful tools

Testing code

Telescope - unit testing

Busted - unit testing and mocking

Lust - unit testing and mocking

Lunit - unit testing

LuaCov - code coverage

Cucumber - BDD

Static code analysis

LuaCheck - used in several editors

LuaInspect


Thank you