Kurs języka Lua

06 – Wstęp do programowania obiektowego

Jakub Kowalski

Instytut Informatyki,
Uniwersytet Wrocławski

2017

Plan na dzisiaj

Opowiemy sobie dokładniej o

1. Modułach
2. Metatabelach
3. Podstawowej obiektowości

Moduły

Moduły w Lua

Ponieważ ograniczona jest cierpliwość programisty zmuszonego do pisania kodu w jednym pliku...
czas się zaprzyjaźnić z modułami.

- Moduł w Lua to zazwyczaj tabela (a jakże) zawierająca wszystkie funkcje, stałe i struktury które ten moduł udostępnia
- Z punktu widzenia organizacji przestrzeni moduł to plik z kodem Lua o odpowiedniej nazwie i w odpowiednim miejscu
- (ewentualnie dll/so, ale o tym kiedyś)
- Program ładuje moduły przy użyciu wbudowanej funkcji `require`, która bierze nazwę modułu i zwraca ten moduł – tak więc zazwyczaj musimy go od razu przypisać do jakiejś zmiennej
- Znamy już moduły biblioteki standardowej, które są domyślnie ładowane do zmiennych globalnych o odpowiednich nazwach

```
math = require('math')
string = require'string'
...
```

Funkcja require

- require jest zwykłą funkcją, bez żadnych dodatkowych przywilejów
- Wczytanym modułem możemy manipulować jak normalną tablicą

```
local mod = require 'mod'
mod.foo()
<-->
local m = require 'mod'
local f = m.foo
f()
<-->
local f = require 'mod'.foo
f()
```

- Lua nie narzuca prawie żadnych ograniczeń na to czym moduł jest
- W szczególności może zwracać coś innego niż tabela, lub mieć skutki uboczne w czasie jego wczytania

Funkcja `require`

- Na początku `require` sprawdza czy moduł nie jest już załadowany w tabeli `package.loaded`
- Jeśli tak to zwraca jego wartość (moduł nie jest ponownie wczytywany)
- W przeciwnym przypadku szuka odpowiedniego pliku korzystając z wytycznych w `package.path` i wczytuje go funkcją `loadfile`
- Jeśli to się nie powiedzie to korzystając z `package.cpath` szuka biblioteki w C którą ładuje funkcją `package.loadlib`
- W tym momencie `require` ma do dyspozycji *loader*, czyli funkcję która po wywołaniu zwróci moduł
- Ostatecznie, `require` wywołuje loader z dwoma argumentami (dostępnymi w module za pośrednictwem `...`): nazwą modułu oraz ścieżką do pliku który został wczytany
- Wartość zwrócona przez loader zostaje zachowana w `package.loaded` na poczet przyszłych wywołań

Definiowanie modułów

```
local mymodule = {}
function mymodule.foo() print("Hello World!") end
mymodule.const = 667
return mymodule
```

```
local function foo() print("Hello World!") end
local constant = 667
return {foo = foo, const=constant}
```

```
local mymodule = {}
package.loaded[...] = mymodule
function mymodule.foo() print("Hello World!") end
mymodule.const = 667
```

Definiowanie modułów – prywatność

```
local mymodule = {}
local function phello(s) print ('Hello '..s) end
function mymodule.foo() phello('World') end
mymodule.const = 667
return mymodule
```

```
local function phello(s) print ('Hello '..s) end
local function foo() phello('World') end
local constant = 667
return {foo = foo, const=constant}
```

Scieżka poszukiwań

package.path

- zawiera ścieżki pod którymi będą szukane moduły

```
;I:\ZeroBraneStudio\bin\lua\?.lua; ...
I:\ZeroBraneStudio\bin\..\share\lua\5.3\?.lua;
.\?.lua;.\?\init.lua;./?.lua;./?/init.lua;...
```

- znak zapytania zamienia się na nazwę modułu
- znak kropki w nazwie modułu zamienia się na separator folderów
- tak więc `require ¨a.b¨` będzie próbowało:
  - `a/b.lua`
  - `a/b/init.lua`
  - ...

package.searchers

- zawiera funkcje które zwracają loadery
- argumentem takiej funkcji jest nazwa modułu
- system wspomaga customizację, np. gdybyśmy chcieli trzymać spakowane moduły – wystarczy dodać do listy odpowiednią funkcję

Przydatne tricki

Wymuszenie ponownego ładowania modułu

```
package.loaded.<modname> = nil
local mod = require "<modname>"
```

Incjalizacja modułu

- Nie istnieje mechanizm inicjalizowania modułu za pośrednictwem funkcji `require`.
- Aby to w miarę przyjaźnie zasymulować, najproście, jest stworzyć w module funkcję która go inicjalizuje i zwraca

W module:

```
function module.init(a, b, c)
  <inicjalizacja>
  return module
end
```

W pliku głównym:

```
local mod = require "module".init(1, 2, 3)
```

Przydatne tricki

Argumenty modułu

W module:

```
for k, v in ipairs{...} do print (k, v) end
```

W pliku głównym:

```
local m = require "a.b"
--> 1   a.b
--> 2   .\a\b\init.lua
```

Proste symulowanie require

```
local m = require <modulename>
<~~>
local m = (function ()
  <modulecontent>
end)()
```

Metatabele

Metatabele

Metatabela, to specjalna tabela która pozwala modyfikować zachowanie innej tabeli, i np.:

- używać arytmetyki, konkatenacji, operatorów relacyjnych
- nadpisać zachowanie ==, ~= i #
- nadpisać zachowanie wbudowanych metod `tostring`, `pairs`, `ipairs`.
- przechwytywać zapytania do nieistniejących pól i tworzenie nowych pól
- wywołać tablicę jak funkcję

```
t = {}
print(getmetatable(t)) --> nil
t1 = {}
setmetatable(t, t1) -- error jesli istnieje
print(getmetatable(t) == t1) --> true
```

Metatabele

- każda tabela może mieć jedną metatabelę która będzie modyfikowała jej zachowanie
- wiele tabel może współdzielić jedną metatabelę
- można być swoją własną metatabelą
  - tabele te będą się więc podobnie zachowywały
  - w ten sposób definiujemy typy danych

Metatabele nie-tabel

- z poziomu Lua możemy modyfikować jedynie metatabele tabel
- metatabelami innych typów można manipulować z poziomu C
- poza napisami typy generalnie nie mają przypisanych metatabel

```
print(getmetatable(10))     --> nil
print(getmetatable(print)) --> nil
```

- metatabela dla napisów jest ustawiona przez bibliotekę `string`

```
print(getmetatable('ala')) --> table: 007a8420
print(getmetatable('kot')) --> table: 007a8420
```

Metatabele

- każda tabela może mieć jedną metatabelę która będzie modyfikowała jej zachowanie
- wiele tabel może współdzielić jedną metatabelę
- można być swoją własną metatabelą
  - tabele te będą się więc podobnie zachowywały
  - w ten sposób definiujemy typy danych

Metatabele nie-tabel

- z poziomu Lua możemy modyfikować jedynie metatabele tabel
- metatabelami innych typów można manipulować z poziomu C
- poza napisami typy generalnie nie mają przypisanych metatabel

```
print(getmetatable(10))     --> nil
print(getmetatable(print)) --> nil
```

- metatabela dla napisów jest ustawiona przez bibliotekę `string`

```
function string.test(s) print ('test'..s) end
('code'):test() --> testcode
```

Metametody

- operacje które metatabela modyfikuje ustalamy poprzez pisanie *metametod*
- metametoda to funkcja o specjalnej nazwie:

```
__add, __sub, __mul, __div, __mod,
__pow, __unm, __idiv,
__band, __bor, __bxor, __bnot, __shl, __shr
  -- operacje boolowskie
__eq, __lt, __le,
__concat, __len,  __index, __newindex,
__call, __tostring, __ipairs, __pairs,
__mode, __gc
```

- W przypadku `__index` i `__newindex` mogą to byc również tablice (co jest podstawą systemu obiektowego)

Przykład – zbiory

```
Set = {}
function Set.new (l)
  local set = {}
  for _, v in ipairs(l) 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 res[k] = b[k] end
  return res
end
```

Przykład – zbiory

Moduł Set

```
Set = {}
-- ...
function Set.tostring (set)
  local l = {}
  for e in pairs(set) do
    l[#l+1] = tostring(e)
  end
  return "{" .. table.concat(l, ",") .. "}"
end
function Set.print (s)
  print(Set.tostring(s))
end

return Set
```

Metametody arytmetyczne

Nowa wersja metody Set.new

```
Set = {}
Set.mt = {} -- tworzymy metatabelę
function Set.new (t)
  local set = {}
  setmetatable(set, Set.mt)
  for _, l in ipairs(t) do set[l] = true end
  return set
end
```

Operator dodawania jako suma zbiorów

```
Set.mt.__add = Set.union
Set.print(Set.new{10,20,30} + Set.new{1,30,36})
--> {1,20,36,30,10}
```

Metametody arytmetyczne

Operator mnożenia jako iloczyn zbiorów

```
Set.mt.__mul = Set.intersection
Set.print(Set.new{10,20,30} * Set.new{1,30,36})
--> {30}
```

A co gdy spróbujemy dodać nie zbiór?

```
Set.print(128 + Set.new{10,20,30})
--> bad argument #1 to 'pairs'
 (table expected, got number)
```

Kolejność działań:

- Jeśli pierwszy argument ma odpowiednią metametodę jest ona używana
- Jeśli drugi argument ma odpowiednią metametodę jest ona używana
- W przeciwnym wypadku wywoływany jest błąd

Metametody arytmetyczne

Jak to poprawić?

```
function Set.union (a,b)
  if getmetatable(a) ~= Set.mt or
     getmetatable(b) ~= Set.mt then
       error("Cannot add set to a non-set", 2)
  <jak wcześniej>
end
```

Albo

```
function Set.union (a,b)
  if type(a) == 'number' then a = Set.new{a} end
  if type(b) == 'number' then b = Set.new{b} end
  <dodatkowe warunki na error>
  <reszta jak wcześniej>
end
```

Metametody relacyjne

- Porównanie wartości różnych typów zawsze jest fałszem!
- Można nadpisać jedynie operatory ==, <, <=
  - a~=b jest obliczane jako not(a==b)
  - a>b jest obliczane jako b<a, zaś a=>b jest obliczane jako b<=a

```
Set.mt.__le = function (a,b) -- "<="
  for k in pairs(a) do
    if not b[k] then return false end
  end
  return true
end
Set.mt.__lt = function (a,b) -- "<"
  return a <= b and not (b <= a)
end
Set.mt.__eq = function (a,b) -- "="
  return a <= b and b <= a
end
```

Metametody biblioteczne

tostring

```
s1 = Set.new{4, 10, 2}
print (s1) --> table: 0067c0b0
Set.mt.__tostring = Set.tostring
print (s1) --> {4,10,2}
```

Ochrona dostępu

Nadpisanie metametody __metatable zabezpiecza przed podejrzeniem oraz modyfikacją metatabeli przez użytkownika.

```
Set.mt.__metatable = "not your business"
s1 = Set.new{}
print(getmetatable(s1)) --> not your business
setmetatable(s1, {})
-->  cannot change protected metatable
```

Metametody tablicowe

- Jeśli pole w tablicy nie istnieje, zwracany jest `nil`
- Chyba, że metatablica posiada metodę `__index`
- Dzięki temu potrafimy definiować dziedziczenie, a więc obiekty oparte na tym samym prototypie
- Jeśli drugi argument ma odpowiednią metametodę jest ona używana
- W przeciwnym wypadku wywoływany jest błąd

```
prototype = {x=0, y=0, width=100, height=100}
local mt = {} -- metatable
function new (o) -- konstruktor
  setmetatable (o, mt)
  return o
end
mt.__index = function (_, key)
  return prototype[key]
end
w = new{x=10, y=20}
print (w.width) --> 100
```

Metametody tablicowe

Jak to działa:

- Lua wykrywa, że danego pola nie ma w tablicy,
- ale istnieje metametabela z metodą `__index`
- Jej argumentami są tablica `w` i nieobecny klucz `'width'`
- Funkcja odpytuje więc prototyp i zwraca odpowiednią wartość.

Wykorzystywanie `__index` do dziedziczenia jest tak powszechne, że stworzono skrót, pozwalający na nadpisanie tej metametody tablicą. (co jest również szybsze)

```
mt.__index = prototype -- działa tak samo
```

Bezpośredni dostęp

Aby uzyskać dostęp do tablicy bez wywoływania metametody, należy użyć funkcji `rawget(t, i)`. (dostęp tą metodą nie jest szybszy)

```
print (rawget(w, 'width')) --> nil
```

Metametody tablicowe

- Metametoda `__newindex` działa dla przypisania tak jak `__index` dla odczytu
- Jeśli próbujemy dokonać zapisu do nieistniejącego pola tablicy, interpreter najpierw sprawdza `__newindex`
- W postaci funkcyjnej metoda dostaje trzy argumenty: `__newindex(table, key, value)`
- Metoda `__newindex` także może być zastępiona przez tablicę – wtedy zapis jest wykonywany na tablicy wskazywanej przez „metodę".

Bezpośredni dostęp

Wywołanie metametody można ominąć dokonując zapisu za pośrednictwem funkcji `rawset`.

```
rawset(t,k,v) <--> t[k]=v
```

Przykład: wartości domyślne

Za pomocą metametod, możemy zmienić domyślną wartość (`nil`) pola.

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

tab = {x=10, y=20}
print (tab.x, tab.z) --> 10    nil
setDefault (tab, 0)
print (tab.x, tab.z) --> 10    0
```

- Funkcja `setDefault` tworzy nowe domknięcie i metatablicę dla każdej tablicy dla której ją wywołamy
- Nie możemy użyć jednej metatablicy do obsługi tablic o różnych wartościach domyślnych
- Ale chcielibyśmy to zmienić i móc ustawiać różne wartości domyślne nie tworząc nowych metatablic

Przykład: wartości domyślne

Jeśli nie obawiamy się konfliktu nazw

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

Jeśli obawiamy się konfliktu nazw

```
local unique_key = {}   -- unikatowy klucz
local mt = {__index =
      function (t) return t[unique_key] end}
function setDefault (t, d)
  t[unique_key] = d
  setmetatable(t, mt)
end
```

Przykład: śledzenie dostępu

```
function track (t)
-- index/newindex działają tylko na pustych polach
  local proxy = {}
  local mt = {
    __index = function (_, k)
      print("*access to element " .. tostring(k))
      return t[k]
    end,
    __newindex = function (_,k,v)
      print("*update of element " .. tostring(k) .. "
      t[k] = v
    end,
  }
  setmetatable(proxy, mt)
  return proxy
end
```

Przykład: śledzenie dostępu

```
t = {}
t = track(t)
t[2] = 'hello'
--> *update of element 2 to hello
print(t[2])
--> *access to element 2
--> hello
```

```
t = track{10, 20}
print (#t) --> 2
for k, v in pairs(t) do print(k, v) end
--> *traversing element 1
--> 1  10
--> *traversing element 2
--> 2  20
```

Przykład: śledzenie dostępu

```
...
    __pairs = function ()
      return function (_, k)
        local nk, nv = next(t, k)
        if nk ~= nil then
          print("*traversing element "
                .. tostring(nk))
        end
        return nk, nv
      end
    end,
    __len = function () return #t end
...
```

Podobnie jak wcześniej możemy używać jednej metatabeli w której mapujemy różne proxy na ich oryginalne tabele.

Przykład: tablice tylko do odczytu

```
function readOnly (t)
  local proxy = {}
  local mt = {
    __index = t,
    __newindex = function (t,k,v)
      error("updating read-only table", 2)
    end
  }
  setmetatable(proxy, mt)
  return proxy
end

days = readOnly{"Sunday", "Monday", "Tuesday",
  "Wednesday", "Thursday", "Friday", "Saturday"}
print(days[1]) --> Sunday
days[2] = "Noday"
--> updating read-only table
```

Obiekty

Tablice a obiekty

Tablice w Lua

- maja swój stan oraz „tożsamość” niezależną od aktualnej wartości,
- w różnym czasie mogą przyjmować różne wartości
- mogą mieć przypisane swoje operacje

```
Account = {balance = 0}
function Account.withdraw (v)
  Account.balance = Account.balance - v
end
Account.withdraw(100.00)
```

Działa to prawie jak metoda...

- ponieważ wewnątrz funkcji używamy nazwy globalnej, funkcja będzie działała tylko dla tego obiektu;
- nawet dla tego obiektu, funkcja wymaga żeby był on trzymany w konkretnej zmiennej globalnej.

```
a, Account = Account, nil;      a.withdraw(100.0)
```

Tablice jako obiekty

- To czego nam brakuje to wskazania na obiekt, którego metoda dotyczy.
- Dodajmy więc parametr pełniący rolę `self`/`this`

```
Account = {balance = 0}
function Account.withdraw (self, v)
  self.balance = self.balance - v
end

a1 = { balance = 0; withdraw = Account.withdraw }
a1.withdraw(a1, 100.00)
a2 = { balance = 0; withdraw = Account.withdraw }
a2.withdraw(a2, 260.00)
```

Tablice jako obiekty

- Definiowanie parametru `self` metod, w większości języków obiektowych jest ukryte
- W Lua możemy skorzystać z operatora dwukropka

```
Account = {balance = 0}
function Account:withdraw(v)
  self.balance = self.balance - v
end

a1 = { balance = 0; withdraw = Account.withdraw }
a1:withdraw(100.00)
a2 = { balance = 0; withdraw = Account.withdraw }
a2:withdraw(260.00)
```

Tablice jako obiekty

- Dwukropek jest jedynie skrótem syntaktycznym dodającym dodatkowy argument `self`
- Obie notacje można dowolnie mieszać.

```
Account = {
  balance=0,
  withdraw = function (self, v)
               self.balance = self.balance - v
             end
}

function Account:deposit (v)
  self.balance = self.balance + v
end
Account.deposit(Account, 200.00)
Account:withdraw(100.00)
```

Klasy

- To czego nam brakuje to systemu klas, dziedziczenia oraz prywatności
- Zajmijmy się pierwszą kwestią:
  Jak tworzyć obiekty o podobnym zachowaniu?
- W Lua system klas jest tak naprawdę systemem opartym na prototypach:
  obiekt może odwołać się do innego obiektu (swojego prototypu) w celu wywołania każdej operacji której nie ma zdefiniowanej.
- Do tworzenia prototypów/dziedziczenia używamy oczywiście metatabel i metametod.
- Aby obiekt `B` był prototypem obiektu `A` wystarczy

```
setmetatable(A, {__index = B})
```

  - `A` korzysta z operacji zapisanych w `B` jeśli sam ich nie posiada.
- Określenie, że `B` jest klasą obiektu `A` to jedynie zmiana terminologii.

Klasy

- Chcemy móc tworzyć inne obiekty zachowujące się jak `Account`

```
local mt = {__index = Account}
function Account.new (o)
  o = o or {}
  setmetatable(o, mt)
  return o
end
a = Account.new{balance = 0}
```

- Jak rozumiemy wywołanie:

```
a:deposit(100.00) ?
```

- rozwijamy dwukropek, brak metody odwołuje nas do metatabeli

```
getmetatable(a).__index.deposit(a, 100.00)
```

- ponieważ metatablica to `mt`, a `mt.__index` to `Acoount`, mamy:

```
Account.deposit(a, 100.00)
```

Klasy

- Zamiast tworzyć specjalną metatabelę, możemy użyć samego `Account`
- Możemy użyć dwukropka także do definiowania metody `new`

```
function Account:new (o)
  o = o or {}
  setmetatable(o, self)
  self.__index = self
  return o
end

b = Account:new{balance = 0}
```

- Dziedziczenie działa także dla pól

```
print (b.balance) --> 0
b.balance = b.balance + v
```

- Teraz `b` ma już pole `balance` więc kolejne odwołania do niego nie będą już wywoływały metametody

Klasy

- Alternatywna skrótowa forma – „anonimowa” metatabela:

```
function Account:new (obj)
  obj = obj or {}
  return setmetatable(obj, { __index = self })
end
```

- Dzięki temu nie brudzimy metametodami w przestrzeni samej klasy `Account`
- Funkcja `setmetatable` zwraca swój pierwszy argument

Dziękuję za uwagę

Za tydzień:
Podstawy C API,
może jeszcze trochę o obiektowości,
...?