

# Gli oggetti di Lua

Roberto Giacomelli

Articolo sul blog <http://robitec.wordpress.com>

e-mail: giaconet dot mailbox at gmail dot com

12 ottobre 2011

## Sommario

Il linguaggio Lua è fondato sull'essenzialità tanto che supporta la programmazione ad oggetti utilizzando quasi esclusivamente le risorse di base e senza mettere a disposizione specifici costrutti. In questa introduzione vedremo come ciò sia stato effettivamente implementato utilizzando l'unica struttura dati disponibile in Lua: la *tabella*.

Per la comprensione del testo occorre una conoscenza di base sia di Lua, sia dei concetti del paradigma di programmazione ad oggetti, utile per introdurre nei programmi un più elevato livello d'astrazione.

## Indice

<b>1</b>	<b>Un rettangolo</b>	<b>1</b>
<b>2</b>	<b>Prima classe</b>	<b>2</b>
2.1	Metatabelle . . . . .	2
2.2	Il metametodo <code>__index()</code> . . . . .	3
2.3	Di nuovo un rettangolo . . . . .	3
2.4	Come funziona? . . . . .	3
2.5	Questa volta un cerchio . . . . .	3
2.6	Come funziona? . . . . .	4
<b>3</b>	<b>Conclusioni</b>	<b>4</b>
<b>4</b>	<b>Note tecniche</b>	<b>4</b>
<b>5</b>	<b>Licenza ed informazioni varie</b>	<b>4</b>
5.1	Distribuzione/Citazioni . . . . .	5
5.2	Colophon . . . . .	5

## 1 Un rettangolo

Come esempio costruiremo un oggetto per rappresentare un rettangolo, un ente geometrico definibile con due parametri, le lunghezze dei lati, e dotato di una proprietà calcolabile chiamata *area*. Gli elementi semplici che memorizzano i valori dei lati sono detti *campi* mentre la proprietà dell'area che ne dipende è detta *metodo*.

Gli oggetti in Lua si rappresentano con le tabelle che possono contenere valori come numeri ed anche funzioni. Un primo tentativo potrebbe essere questo (sperimentatelo in modo interattivo al terminale per rendervi meglio conto del codice):

```
-- prima implementazione
Rettangolo = {} -- creazione tabella (oggetto)
```

```
-- creazione di due campi
Rettangolo.a = 12
Rettangolo.b = 7

-- un metodo
function Rettangolo.area ()
    -- accesso alla variabile 'Rettangolo'
    return Rettangolo.a * Rettangolo.b
end

-- primo test
print(Rettangolo.area()) --> stampa 84, OK
print(Rettangolo.a)      --> stampa 12, OK
```

Ci accorgiamo presto che questa implementazione basata sulle tabelle è difettosa in quanto non rispetta l'indipendenza degli oggetti rispetto al loro nome, ed infatti il prossimo test fallisce:

```
-- ancora la prima implementazione
Rettangolo = {a=12, b=7}
```

```

-- un metodo
function Rettangolo.area ( )
  -- accesso alla variabile 'Rettangolo'
  return Rettangolo.a * Rettangolo.b
end

-- secondo test
r = Rettangolo -- creiamo un secondo riferimento
Rettangolo = nil -- distruggiamo il riferimento
                originale

print(r.a)      --> stampa 12, OK
print(r.area()) --> errore!

```

Il problema sta nel fatto che nel metodo `area()` compare un particolare riferimento che invece deve poter essere qualunque. La soluzione è introdurre il riferimento all'oggetto come parametro esplicito nel metodo stesso, che chiameremo `self`, così da poter generalizzarne la validità. Questa idea è quella utilizzata dai linguaggi di programmazione che supportano gli oggetti.

Secondo questo nuovo schema, dovremo riscrivere il metodo `area()` in questo modo:

```

-- seconda implementazione
Rettangolo = {a=12, b=7}

-- il metodo diviene indipendente dal particolare
-- riferimento all'oggetto:
function Rettangolo.area ( self )
  return self.a * self.b
end

-- ed ora il test
myrect = Rettangolo
Rettangolo = nil -- distruggiamo il riferimento

print(myrect.a)      --> stampa 12, OK
print(myrect.area(myrect)) --> stampa 84, OK

```

Fino ad ora abbiamo costruito l'oggetto sfruttando le caratteristiche della tabella e la particolarità che consente di assegnare una funzione ad una variabile, ma da questo momento entra in scena l'operatore due punti (`:`), nella chiamata di funzione permettendo di passare il riferimento implicitamente.

Questo operatore è il primo nuovo elemento di Lua inteso per supportare la programmazione orientata agli oggetti, ed eccone una descrizione:

*Se si accede ad un metodo memorizzato in una tabella con l'operatore due punti `:` anziché con l'operatore `.`, sarà aggiunto implicitamente un primo parametro con il riferimento alla tabella stessa chiamato `self`.*

Le seguenti due espressioni sono perfettamente equivalenti per risultato pratico, ma hanno due differenti punti di vista concettuali:

```

-- operatore punto
-- dobbiamo inserire il riferimento
-- come primo parametro della funzione
print(myrec.area(myrec)) --> stampa 84, OK

-- operatore due punti

```

```

-- il riferimento 'myrec' viene
-- passato implicitamente
print(myrec.area()) --> stampa 84, OK

```

## 2 Prima classe

Il salto definitivo nella programmazione **OO** (Object Oriented Programming) è la costruzione di una *classe*. Dovremo infatti poter costruire nuovi oggetti senza ogni volta assemblare i campi ed i metodi, insomma serve un qualcosa che faccia da stampo.

Per mantenere la filosofia di essenzialità, Lua non implementa, come invece troviamo in altri linguaggi simili, per esempio Python, una nuova parola chiave *class* con cui si definisce un prototipo ma, ancora una volta, propone una soluzione basata sulle tabelle, anzi sulle *metatabelle*.

### 2.1 Metatabelle

Una metatabella è una tabella che contiene funzioni dai nomi prestabiliti che vengono eseguiti quando si verificano particolari eventi, come la richiesta di somma tra due tabelle. Ogni tabella può essere associata ad una propria metatabella e questo consente di creare degli insiemi di tabelle che condividono una stessa aritmetica, giusto per rimanere in nell'esempio precedente.

I nomi di queste funzioni particolari iniziano tutti con un doppio trattino basso, per esempio nel caso della somma sarà richiesta la funzione `__add()` della metatabella associata, e vengono chiamati *metametodi*.

Il meta-metodo più semplice da imparare è `__tostring()`, che viene chiamato quando alla funzione `print()` viene data una tabella. In una sessione di terminale scriviamo:

```

-- un numero complesso
complex = {real = 4, imag=-9}
print(complex) --> stampa: 'table: 0x9eb65a8'

-- un qualcosa di più utile:
function printComplex( c )
  local r = string.format("%0.2f",c.real)
  if c.imag == 0 then -- numero reale
    return ("..r..")
  end
  -- numero complesso
  local i = string.format("%0.2f",c.imag)
  return ("..r..",".i..")
end

-- creo la metatabella
mt = {}
mt.__tostring = printComplex

setmetatable(complex, mt)

-- riprovo la stampa
print(complex) --> stampa '(4.00,-9.00)'
```

## 2.2 Il metametodo `__index()`

Il 'metametodo' che interessa per la OOP in Lua è `__index()`. Esso interviene quando chiamiamo un componente di una tabella che non esiste e che normalmente restituirebbe il valore `nil`.

Ecco un esempio di codice (nell'immagine la relativa sessione al terminale):

```
-- una tabella con un campo 'a'
t = {a = 'Campo A'}

print(t.a) --> stampa 'Campo A'
print(t.b) --> stampa 'nil'
```

Costruiamo invece la funzione `__index()` e vediamo che succede:

```
t = {a='Campo A'}

mt = {} -- la metatabella

-- creazione del metametodo
function idx()
    return 'Attenzione: campo inesistente!'
end

-- inseriamolo nella tabella mt
mt.__index = idx

-- assegnamo 'mt' come metatabella di 't'
setmetatable(t, mt)

-- adesso riproviamo ad accedere al metodo b
print(t.b) --> stampa 'Attenzione: campo inesistente!'
```

## 2.3 Di nuovo un rettangolo

Torniamo adesso all'oggetto `Rettangolo` e creiamo una tabella che fungerà da *classe* per gli oggetti, assegnando campi e metodi. Per creare nuovi oggetti, creeremo un metodo `new()` che esaudirà la richiesta assumendo il ruolo di *costruttore*.

```
-- il nuovo oggetto Rettangolo

-- campi
Rettangolo = {a=10, b=10}

-- metodo
function Rettangolo:area()
    return self.a * self.b
end

-- creazione metametodo
Rettangolo.__index = Rettangolo

-- il costruttore
function Rettangolo:new( o )
    -- creazione nuova tabella
    -- se non ne viene fornita una
    o = o or {}

    -- assegnazione metatabella
    setmetatable(o, self)

    -- restituzione riferimento oggetto
    return o
end

-- test -----
-- rettangolo 10 x 10
r1 = Rettangolo:new()
```

```
print(r1.a) --> stampa 10 OK
print(r1:area()) --> stampa 100 OK

-- rettangolo 200 x 10
r2 = Rettangolo:new{a=200}
print(r2.a) --> stampa 200 OK
print(r2:area()) --> stampa 2000 OK

-- rettangolo 12 x 7
r3 = Rettangolo:new{a=12,b=7}
print(r3:area()) --> stampa 84 OK
```

## 2.4 Come funziona?

Il 'costruttore' accetta una tabella che gli viene passata dall'utente, altrimenti ne crea una vuota e la restituisce non appena ne abbia assegnato la metatabella che non è altro che l'oggetto `Rettangolo` stesso visto che l'operatore due punti passa a `new` il riferimento implicito a `Rettangolo`, per cui `self` punta a `Rettangolo`.

Nel primo caso del test, viene chiamato il campo `a` dell'oggetto `r1` che non esiste, allora Lua, poiché invece esiste una metatabella associata ad `r1`, ne chiama il metodo `__index`, che restituisce semplicemente la tabella `Rettangolo` stessa. Alla fine viene restituito il campo `a` di `Rettangolo` che vale 10.

Stessa cosa succede quando viene chiamata la funzione `area` con la tecnica dell'operatore due punti, ovvero la funzione chiamata è quella del corrispondente campo in `Rettangolo` poiché non esiste un campo simile in `r1`.

Analoghe spiegazioni valgono quando al costruttore viene passata una tabella con un unico campo oppure con tutti e due i campi `a` e `b`. Questa volta l'oggetto possiede effettivamente campi propri per i lati ed il metametodo `__index` non viene considerato.

## 2.5 Questa volta un cerchio

Costruiamo un oggetto `Cerchio` che consenta di aggiungere una quantità al raggio. Ci renderemo meglio conto di come funziona il meccanismo nascosto ed automatico delle metatabelle:

```
Cerchio = {radius=0}
Cerchio.__index = Cerchio

function Cerchio:area()
    return math.pi*self.radius^2
end

function Cerchio:addToRadius(v)
    self.radius = self.radius + v
end

function Cerchio:__toString()
    local frmt = 'Io sono un cerchio di raggio %.2f.'
    return string.format(frmt, self.radius)
end

function Cerchio:new(r)
    -- il costruttore attende l'eventuale
    -- valore del raggio del cerchio
    local o = {}

    if r then
```

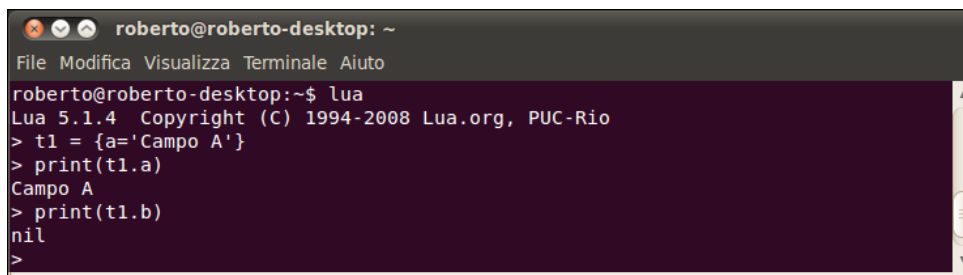


Figura 1: Lua in sessione interattiva al terminale di Ubuntu Linux

```
o.radius=r
end

setmetatable(o, self)
return o
end

-- test -----
o = Cerchio:new()
print(o)

o:addToRadius(12.345)

print(o)
print(o:area())
```

## 2.6 Come funziona?

Vorrei soffermarmi su quanto accade nel codice di test. Abbiamo creato un cerchio senza fornire alcun valore per il raggio. Ciò significa che quando lo stampiamo con la successiva istruzione, il raggio è quello dell'oggetto `Cerchio`, configurato al valore di default nullo, per effetto della chiamata ad `__index` della metatabella.

*Fino a questo momento la tabella dell'oggetto o non contiene alcun campo radius. Cosa succede allora quando viene lanciato il comando `o:addToRadius(12.345)`?*

Il metodo `addToRadius()` contiene una sola espressione. Come da regola viene prima valutata la parte a destra `self.radius + v`, dunque il primo termine assume il valore previsto in `Cerchio` grazie al metametodo, ed alla fine viene creato anche per l'oggetto `o` il campo `radius`.

## 3 Conclusioni

Non tratteremo in questo post il meccanismo peraltro semplice dell'ereditarietà degli oggetti in Lua. Ciò detto possiamo renderci conto che in Lua si possono implementare strutture con il paradigma della OOP. Lo scopo è quello di fornire all'utente funzionalità più facili da utilizzare e contemporaneamente razionalizzare il codice che dovrebbe essere più facile da gestire nel tempo.

Una volta compreso il meccanismo basato sulle metatabelle, possiamo ideare facilmente e con una

certa flessibilità classi di oggetti, ponendo l'attenzione sui metodi e sui campi. In ultimo vi faccio osservare che in Lua non è previsto nessun meccanismo da parte del linguaggio per rendere privati metodi e campi. Scelta coerente per un linguaggio di scripting che mira ad essere essenziale, quindi non ci rimane semplicemente non accedere ai campi, e sarà mantenuta la privacy degli oggetti.

## 4 Note tecniche

Per le note tecniche sottolineo che Lua è un software libero multiplatforma pertanto è liberamente installabile su sistemi meno diffusi e, ovviamente, anche in Windows, Linux e Mac.

Per sapere qualcosa di più sulle tabelle potete consultare i post di questo stesso Blog, in particolare [questo](#) dove trovate un'introduzione alle tabelle di Lua.

Per la comprensione dei meccanismi di Lua per la OOP è caldamente consigliato ripetere gli esempi proposti, eventualmente utilizzando la modalità interattiva al terminale (è sufficiente digitare alla linea di comando `lua` per entrarvi) o lavorando con i file, proponendo altri esempi interessanti (postateli nei commenti mi raccomando...). Happy Lua!

## 5 Licenza ed informazioni varie

Questo articolo come tutto il materiale didattico/divulgativo del blog [robiteX.wordpress.com](http://robiteX.wordpress.com) è rilasciato sotto licenza Creative Commons "Attribuzione-Non commerciale-Non opere derivate" 2.5 Italia, il cui testo integrale con valore legale è consultabile a [questo indirizzo](#). Ciò significa che:

1. Bisogna sempre attribuire la paternità del materiale a <http://robiteX.wordpress.com>;
2. Non si può usare il materiale per fini commerciali;
3. Non si può alterare o trasformare i contenuti, né usarne stralci per creare altre opere.

Se esplicitamente indicato nei commenti iniziali, il codice relativo a programmi software è rilasciato nella specifica licenza.

## 5.1 Distribuzione/Citazioni

Ogni volta che usi o distribuisce parti redistribuibili quest'opera devi farlo secondo i termini con cui esse sono state rilasciate e avendo cura di comunicare tali termini con chiarezza. Ricorda di inserire sempre un hyperlink alla risorsa che redistribuisci o citi.

Il modo migliore per dimostrarmi il vostro apprezzamento è semplicemente quello di linkare direttamente le pagine del blog, senza copiare gli articoli in altri siti, oltre naturalmente a lasciare un commento. Ci sono però casi in cui vorreste poter estrapolare alcune parole dai miei articoli per incuriosire i vostri lettori. In quel caso la prassi convenuta e che, grazie a tutti i blogger seri, viene coscienziosamente rispettata, è questa:

- Creare un “blockquote”, ossia un campo in cui è inserita una citazione;

- Inserire nel blockquote solo il primo periodo (le prime poche frasi) di un articolo, diciamo fino ad arrivare al link “Leggi il resto...”;
- Linkare la rimanente parte dell'articolo all'originale su <http://robotex.wordpress.com>.

Grazie per la collaborazione.

## 5.2 Colophon

Questo documento è stato composto con  $\text{\LaTeX}$  attraverso uno script in Lua chiamato `wp2pdf` che elabora il file originale *html* del post pubblicato sul blog in WordPress. Si tratta di una versione migliorata del codice pubblicato sul blog stesso. Per saperne di più contattatemi via posta elettronica all'indirizzo nel titolo del documento, o lasciate un commento sul blog.