Metamethods Tutorial

This is a brief introduction to the concepts of Lua metamethods. Once you have read this you might want to read further examples of metamethod use, for example to implement classes for object oriented programming, in ObjectOrientationTutorial and LuaClassesWithMetatable. Section 2.8 covers Metatables in the Reference Manual.

Metamethods

Lua has a powerful extension mechanism which allows you to overload certain operations on Lua objects. Only tables and userdata objects can use this functionality. Each overloaded object has a *metatable* of function *metamethods* associated with it; these are called when appropriate, just like operator overloads in C++. Unlike C++ though you can modify the metamethods associated with an object at runtime.

The metatable is a regular Lua table containing a set of metamethods, which are associated with *events* in Lua. Events occur when Lua executes certain operations, like addition, string concatenation, comparisons etc. Metamethods are regular Lua functions which are called when a specific event occurs. The events have names like "add" and "concat" (see manual section 2.8) which correspond with metamethods with names like "`__add`" and "`__concat`". In this case to add or concatenate two Lua objects. These are defined as usual in a table in key-value pairs.

Metatables

We use the function `setmetatable()` to associate a metatable with an appropriate object. Let's have an example:

```
> x = { value=3 }      -- our object
>
> mt = { __add = function (a,b)
>>                  return { value = a.value + b.value }
>>                end }  -- metatable containing event callbacks
>
> a = x+x              -- without a metatable this is just a regular table
stdin:1: attempt to perform arithmetic on global `x' (a table value)
stack traceback:
        stdin:1: in main chunk
        [C]: ?
>
> setmetatable(x, mt)  -- attach our metamethods to our object
>
```

```
> a = x + x            -- try again
> print(a.value)
6
```

In the above example we create a table called `x` containing a value, 3. We create a metatable, containing the event overloads we would like to attach to the table, in `mt`. We are overloading the "add" event here; notice how the function receives two arguments because "add" is a binary operation. We attach the metatable `mt` to the table `x` and when we apply the addition operator to x (in `a = x+x`) we can see that `a` contains the results of the `__add` metamethod.

Notice however that `a` is not an instance of our "class", it is just a plain table with no metamethod associated with it.

```
> b = a+a
stdin:1: attempt to perform arithmetic on global `a' (a table value)
stack traceback:
        stdin:1: in main chunk
        [C]: ?
```

However, as long as an object with the appropriate metamethod is part of the event operation Lua will behave properly. It does not matter which side of the `+` operator our class is as Lua will resolve this.

```
> b = a+x
> print(b.value)
9
```

We can retrieve the metatable from an object with metamethods using `getmetatable(object)`:

```
> = getmetatable(x)
table: 0035BC98
> = getmetatable(x).__add    -- get the "__add" metamethod
function: 0035C040
```

Note, we could attach metamethods to the returned "class" in the above example by doing the following:

```
> x = { value = 3 }
>
> mt = { __add = function (a,b)
>>       return setmetatable({ value = a.value + b.value }, mt)
>>     end }  -- metatable
>
> setmetatable(x, mt)
>
> a = x+x
```

```
> print(a.value)  -- as before
6
> b = a+a
> print(b.value)  -- no error this time as "a" is the same "class" as "x"
12
```

A note on "classes"

Without the metatable attached, the table `x` could be likened to a structure (`struct`) in C. With the ability to overload operations that occur on the table we might say that the table `x` become analogous to a class instance in C++. The reason why we're being cautious to call this a class is that we can do things with this "class instance" that we cannot do in C++, e.g. attach more event overloads, or change overloads at runtime, or manipulate the contents of the "class" dynamically (e.g. add or remove elements).

String class example

The following is a very simple string class. We define a constructor class `String` to return a table, containing our string, with the metatable attached. We define a couple of functions to concatenate strings together and multiply strings.

```
> mt = {}  -- metatable
>
> function String(s)
>>   return setmetatable({ value = s or '' }, mt)
>> end
>
> function mt.__add (a,b)
>>   return String(a.value .. b.value)
>> end
>
> function mt.__mul (a,b)
>>   return String(string.rep(a.value, b))
>> end
>
> s = String('hello ')
>
> print(s.value)
hello
> print( (s + String('Lua user')).value )  -- concat 2 String instances
hello Lua user
> print( (s*3).value )
hello hello hello
> print( (((s + String('Lua user.')))*2).value )  -- use both metamethods
```

```
hello Lua user.hello Lua user.
```

Note that the metamethods are added to the metatable `mt` after it has been referenced by the `String()` constructor. The metatable is dynamic and can be altered even after instances of the String have been created. When metamethods are added or altered this will affect all objects (in this case Strings) with the same metatable instantaneously.

More events

The following are notes on other of the metamethod events that Lua handles.

__index

Two interesting metamethods are `__index` and `__newindex`. When we use the `+` operator Lua automatically associates this with the `__add` event. If the key that we are looking for is not a built in one we can use the `__index` event to catch look ups. This event is called whenever we are looking for a key associated with an object (and it's not one of the built in ones). For example, what it we want to get the length of a string in the above String class example. We could call the string length function, but what if we wanted to treat the length as a number entry in the String instance. Following on from the previous example:

```
> print (s.length)  -- no such key in the String table class
nil
> mt.__index = function (t,key)
>>   if key=='length' then return string.len(t.value) end
>> end
> print (s.length)  -- new __index event calls the above function
6
```

We attach an `__index` event to the String class's metatable. The index metamethod looks for the key "length" and returns the length of the String.

__newindex

Now suppose that we want to make sure that The "value" member remains the only field in the String class. We can do this using the `__newindex` metamethod. This method is called whenever we try to create a new key in the table, not look up an existing one. E.g.,

```
> print (s.value, s.length)
hello   6
> mt.__newindex = function (t,key,value)
>>   error('sorry, the only member is "value"')
>> end
> s.banana = 'yellow'
stdin:2: sorry, the only member is "value"
```

```
stack traceback:
        [C]: in function `error'
        stdin:2: in function <stdin:1>
        stdin:1: in main chunk
        [C]: ?
> s.value = 'abc'  -- no error here
```

Here we just call the Lua `error()` function to create an error whenever the `__newindex` event is invoked.

FindPage · RecentChanges · preferences
edit · history
Last edited December 27, 2003 1:48 am PDT (diff)