LUA - COROUTINES

Introduction

Coroutines are collaborative in nature, which allows two or more methods to execute in a controlled manner. With coroutines, at any given time, only one coroutine runs and this running coroutine only suspends its execution when it explicitly requests to be suspended.

The above definition may look vague. Let us assume we have two methods, one the main program method and a coroutine. When we call a coroutine using resume function, its starts executing and when we call yield function, it suspends executing. Again the same coroutine can continue executing with another resume function call from where it was suspended. This process can continue till the end of execution of the coroutine.

Functions Available in Coroutines

The following table lists all the available functions for coroutines in Lua and their corresponding use.

S.N.	Method & Purpose
1.	**coroutine.create** *f* Creates a new coroutine with a function f and returns an object of type "thread".
2.	**coroutine.resume** *co*[, *val*1, . . .] Resumes the coroutine co and passes the parameters if any. It returns the status of operation and optional other return values.
3.	**coroutine.running** Returns the running coroutine or nil if called in the main thread.
4.	**coroutine.status** *co* Returns one of the values from running, normal, suspended or dead based on the state of the coroutine.
5.	**coroutine.wrap** *f* Like coroutine.create, the coroutine.wrap function also creates a coroutine, but instead of returning the coroutine itself, it returns a function that, when called, resumes the coroutine.
6.	**coroutine.yield . . .** Suspends the running coroutine. The parameter passed to this method acts as additional return values to the resume function.

Example

Let's look at an example to understand the concept of coroutines.

```
co = coroutine.create(function (value1,value2)
   local tempvar3 =10
   print("coroutine section 1", value1, value2, tempvar3)

   local tempvar1 = coroutine.yield(value1+1,value2+1)
   tempvar3 = tempvar3 + value1
   print("coroutine section 2",tempvar1 ,tempvar2, tempvar3)

   local tempvar1, tempvar2= coroutine.yield(value1+value2, value1-value2)
   tempvar3 = tempvar3 + value1
   print("coroutine section 3",tempvar1,tempvar2, tempvar3)
   return value2, "end"

end)

print("main", coroutine.resume(co, 3, 2))
print("main", coroutine.resume(co, 12,14))
print("main", coroutine.resume(co, 5, 6))
print("main", coroutine.resume(co, 10, 20))
```

When we run the above program, we will get the following output.

```
coroutine section 1 3 2 10
main true 4 3
coroutine section 2 12 nil 13
main true 5 1
coroutine section 3 5 6 16
main true 2 end
main false cannot resume dead coroutine
```

What Does the Above Example Do?

As mentioned before, we use the resume function to start the operation and yield function to stop the operation. Also, you can see that there are multiple return values received by resume function of coroutine.

- First, we create a coroutine and assign it to a variable name co and the coroutine takes in two variables as its parameters.
- When we call the first resume function, the values 3 and 2 are retained in the temporary variables value1 and value2 till the end of the coroutine.
- To make you understand this, we have used a tempvar3, which is 10 initially and it gets updated to 13 and 16 by the subsequent calls of the coroutines since value1 is retained as 3 throughout the execution of the coroutine.
- The first coroutine.yield returns two values 4 and 3 to the resume function, which we get by updating the input params 3 and 2 in the yield statement. It also receives the true/false status of coroutine execution.
- Another thing about coroutines is how the next params of resume call is taken care of, in the above example; you can see that the variable the coroutine.yield receives the next call params which provides a powerful way of doing new operation with the retentionship of existing param values.
- Finally, once all the statements in the coroutines are executed, the subsequent calls will return in false and "cannot resume dead coroutine" statement as response.

Another Coroutine Example

Let us look at a simple coroutine that returns a number from 1 to 5 with the help of yield function and resume function. It creates coroutine if not available or else resumes the existing coroutine.

```
function getNumber()
   local function getNumberHelper()
      co = coroutine.create(function ()
      coroutine.yield(1)
      coroutine.yield(2)
      coroutine.yield(3)
      coroutine.yield(4)
      coroutine.yield(5)
      end)
      return co
   end

   if(numberHelper) then
      status, number = coroutine.resume(numberHelper);

      if coroutine.status(numberHelper) == "dead" then
         numberHelper = getNumberHelper()
         status, number = coroutine.resume(numberHelper);
      end

      return number
   else
      numberHelper = getNumberHelper()
      status, number = coroutine.resume(numberHelper);
      return number
   end

end

for index = 1, 10 do
   print(index, getNumber())
end
```

When we run the above program, we will get the following output.

```
1 1
2 2
3 3
4 4
5 5
6 1
7 2
8 3
9 4
10 5
```

There is often a comparison of coroutines with the threads of multiprogramming languages, but we need to understand that coroutines have similar features of thread but they execute only one at a time and never execute concurrently.

We control the program execution sequence to meet the needs with the provision of retaining certain information temporarily. Using global variables with coroutines provides even more flexibility to coroutines.