LUA - ITERATORS

Iterator is a construct that enables you to traverse through the elements of the so called collection or container. In Lua, these collections often refer to tables, which are used to create various data structures like array.

Generic For Iterator

A generic *for* iterator provides the key value pairs of each element in the collection. A simple example is given below.

```
array = {"Lua", "Tutorial"}

for key,value in ipairs(array)
do
   print(key, value)
end
```

When we run the above code, we will get the following output −

```
1  Lua
2  Tutorial
```

The above example uses the default *ipairs* iterator function provided by Lua.

In Lua we use functions to represent iterators. Based on the state maintenance in these iterator functions, we have two main types −

- Stateless Iterators
- Stateful Iterators

Stateless Iterators

By the name itself we can understand that this type of iterator function does not retain any state.

Let us now see an example of creating our own iterator using a simple function that prints the squares of **n** numbers.

```
function square(iteratorMaxCount,currentNumber)

   if currentNumber<iteratorMaxCount
   then
      currentNumber = currentNumber+1
      return currentNumber, currentNumber*currentNumber
   end

end

for i,n in square,3,0
do
   print(i,n)
end
```

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

```
1 1
2 4
3 9
```

The above code can be modified slightly to mimic the way *ipairs* function of iterators work. It is shown below.

```
function square(iteratorMaxCount,currentNumber)

   if currentNumber<iteratorMaxCount
   then
      currentNumber = currentNumber+1
      return currentNumber, currentNumber*currentNumber
   end

end

function squares(iteratorMaxCount)
   return square,iteratorMaxCount,0
end

for i,n in squares(3)
do
   print(i,n)
end
```

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

```
1 1
2 4
3 9
```

Stateful Iterators

The previous example of iteration using function does not retain the state. Each time the function is called, it returns the next element of the collection based on a second variable sent to the function. To hold the state of the current element, closures are used. Closure retain variables values across functions calls. To create a new closure, we create two functions including the closure itself and a factory, the function that creates the closure.

Let us now see an example of creating our own iterator in which we will be using closures.

```
array = {"Lua", "Tutorial"}

function elementIterator (collection)

   local index = 0
   local count = #collection

   -- The closure function is returned

   return function ()
      index = index + 1

      if index <= count
      then
         -- return the current element of the iterator
         return collection[index]
      end

   end

end

for element in elementIterator(array)
do
   print(element)
end
```

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

```
Lua
Tutorial
```

In the above example, we can see that elementIterator has another method inside that uses the local external variables index and count to return each of the element in the collection by incrementing the index each time the function is called.

We can create our own function iterators using closure as shown above and it can return multiple elements for each of the time we iterate through the collection.