LUA - OBJECT ORIENTED

Introduction to OOP

Object Oriented Programming*OOP*, is one the most used programming technique that is used in the modern era of programming. There are a number of programming languages that support OOP which include,

- C++
- Java
- Objective-C
- Smalltalk
- C#
- Ruby

Features of OOP

- **Class** − A class is an extensible template for creating objects, providing initial values for state *membervariables* and implementations of behavior.
- **Objects** − It is an instance of class and has separate memory allocated for itself.
- **Inheritance** − It is a concept by which variables and functions of one class is inherited by another class.
- **Encapsulation** − It is the process of combining the data and functions inside a class. Data can be accessed outside the class with the help of functions. It is also known as data abstraction.

OOP in Lua

You can implement object orientation in Lua with the help of tables and first class functions of Lua. By placing functions and related data into a table, an object is formed. Inheritance can be implemented with the help of metatables, providing a look up mechanism for nonexistent functions*methods* and fields in parent objects.

Tables in Lua have the features of object like state and identity that is independent of its values. Two objects *tables* with the same value are different objects, whereas an object can have different values at different times, but it is always the same object. Like objects, tables have a life cycle that is independent of who created them or where they were created.

A Real World Example

The concept of object orientation is widely used but you need to understand it clearly for proper and maximum benefit.

Let us consider a simple math example. We often encounter situations where we work on different shapes like circle, rectangle and square.

The shapes can have a common property Area. So, we can extend other shapes from the base object shape with the common property area. Each of the shapes can have its own properties and functions like a rectangle can have properties length, breadth, area as its properties and printArea and calculateArea as its functions.

Creating a Simple Class

A simple class implementation for a rectangle with three properties area, length, and breadth is shown below. It also has a printArea function to print the area calculated.

```
-- Meta class
Rectangle = {area = 0, length = 0, breadth = 0}

-- Derived class method new

function Rectangle:new (o,length,breadth)
   o = o or {}
   setmetatable(o, self)
   self.__index = self
   self.length = length or 0
   self.breadth = breadth or 0
   self.area = length*breadth;
   return o
end

-- Derived class method printArea

function Rectangle:printArea ()
   print("The area of Rectangle is ",self.area)
end
```

Creating an Object

Creating an object is the process of allocating memory for the class instance. Each of the objects has its own memory and share the common class data.

```
r = Rectangle:new(nil,10,20)
```

Accessing Properties

We can access the properties in the class using the dot operator as shown below –

```
print(r.length)
```

Accessing Member Function

You can access a member function using the colon operator with the object as shown below –

```
r:printArea()
```

The memory gets allocated and the initial values are set. The initialization process can be compared to constructors in other object oriented languages. It is nothing but a function that enables setting values as shown above.

Complete Example

Lets look at a complete example using object orientation in Lua.

```
-- Meta class
Shape = {area = 0}

-- Base class method new

function Shape:new (o,side)
   o = o or {}
   setmetatable(o, self)
   self.__index = self
   side = side or 0
   self.area = side*side;
   return o
end

-- Base class method printArea

function Shape:printArea ()
```

```
   print("The area is ",self.area)
end

-- Creating an object
myshape = Shape:new(nil,10)

myshape:printArea()
```

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

```
The area is  100
```

Inheritance in Lua

Inheritance is the process of extending simple base objects like shape to rectangles, squares and so on. It is often used in the real world to share and extend the basic properties and functions.

Let us see a simple class extension. We have a class as shown below.

```
 -- Meta class
Shape = {area = 0}

-- Base class method new

function Shape:new (o,side)
   o = o or {}
   setmetatable(o, self)
   self.__index = self
   side = side or 0
   self.area = side*side;
   return o
end

-- Base class method printArea

function Shape:printArea ()
   print("The area is ",self.area)
end
```

We can extend the shape to a square class as shown below.

```
Square = Shape:new()

-- Derived class method new

function Square:new (o,side)
   o = o or Shape:new(o,side)
   setmetatable(o, self)
   self.__index = self
   return o
end
```

Over-riding Base Functions

We can override the base class functions that is instead of using the function in the base class, derived class can have its own implementation as shown below −

```
-- Derived class method printArea

function Square:printArea ()
   print("The area of square is ",self.area)
end
```

Inheritance Complete Example

We can extend the simple class implementation in Lua as shown above with the help of another new method with the help of metatables. All the member variables and functions of base class are retained in the derived class.

```
 -- Meta class
Shape = {area = 0}

-- Base class method new

function Shape:new (o,side)
   o = o or {}
   setmetatable(o, self)
   self.__index = self
   side = side or 0
   self.area = side*side;
   return o
end

-- Base class method printArea

function Shape:printArea ()
   print("The area is ",self.area)
end

-- Creating an object
myshape = Shape:new(nil,10)
myshape:printArea()

Square = Shape:new()

-- Derived class method new

function Square:new (o,side)
   o = o or Shape:new(o,side)
   setmetatable(o, self)
   self.__index = self
   return o
end

-- Derived class method printArea

function Square:printArea ()
   print("The area of square is ",self.area)
end

-- Creating an object
mysquare = Square:new(nil,10)
mysquare:printArea()

Rectangle = Shape:new()

-- Derived class method new

function Rectangle:new (o,length,breadth)
   o = o or Shape:new(o)
   setmetatable(o, self)
   self.__index = self
   self.area = length * breadth
   return o
end

-- Derived class method printArea

function Rectangle:printArea ()
    print("The area of Rectangle is ",self.area)
end

-- Creating an object
```

```
myrectangle = Rectangle:new(nil,10,20)
myrectangle:printArea()
```

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

```
The area is  100
The area of square is  100
The area of Rectangle is  200
```

In the above example, we have created two derived classes − Rectangle and Square from the base class Square. It is possible to override the functions of the base class in derived class. In this example, the derived class overrides the function printArea.

Loading [MathJax]/jax/output/HTML-CSS/jax.js