# Code as Data: Lua and Object-Orientation as a Case Study

Lua is a scripting language of considerable popularity, mainly used as an embedded language to extend software. Lua feels like and looks like a normal imperative language, yet something about it seems rather pure. Lua is intentionally designed to be lightweight, simple, and consistent.

The creators of Lua describe Lua as preferring *"mechanisms instead of policies"* (Ierusalimschy el al., 2018). One way this is put into practice is through Lua's approach to object-orientation. Lua does not have a `class` keyword, nor a `struct` keyword, nor something equivalent; rather than treating object-orientation as some kind of magic construct in the language, a kind of object-orientation is achieved through existing constructs, which reflects a *code as data* paradigm.

## Primer 1: Tables in Lua

Before explaining the details of object-orientation in Lua, a few constructs in Lua should be explained. The first is Lua's *table*. A table is a built-in data structure, essentially a hash table, but with some surprises. Below is an example declaration of such a table:

```lua
-- Declaration of a table
ratings = {
  alsBurerShack = 5,
  mcDonalds = 2,
  ["Sup dogs"] = 3
}

-- Modification after declaration
ratings["Kipos"] = 3

-- Iterating through the table
for restaurant, rating in pairs(ratings) do
  print(restaurant .. ": " .. rating)
end
```

Tables consist of pairs of *keys* which map to *values*. A key can be any type, but is typically a string or integer. Because Lua uses hashing, each attempted access of a value has a time complexity of `O(1)`. If no value is found, then the access will return `nil`.

The table is actually Lua's *only* built-in data structure. Arrays must be implemented as tables:

```
array = {-5, 8, 20}

--[[ The above is equivalent to:
array = {
  [1] = -5,
  [2] = 8,
  [3] = 20
}
]]

-- Iterating and printing through the elements of the array.
for i = 1, #array do
  print(array[i])
end
```

When unspecified, array indexes start at `1` in Lua. However, what designates the start of an array is entirely arbitrary. In the above example, the index `1` is actually just a key which maps to `-5`. If we wanted, `0` could be used as an "index", and so could negative indices, or even indices of other types (like strings). Lua does not force anything, but at some point the object could no longer functionally be described as an array. What is an array and what isn't array is determined by the programmer rather than Lua itself; this keeps with Lua's goal of mechanisms instead of policies.

## Primer 2: First-Class Functions in Lua

Functions can be declared plainly in Lua as follows:

```
function add(a, b)
  return a + b
end

print(add(5, 3)) -- Prints 8
```

However, they can also be declared in another way:

```
minus = function(a, b)
  return a - b
end

print(minus(5, 3)) -- Prints 2
```

Above, `minus` can be described as a function. However, it can also be said that `minus` is simply another variable, with its value *set to a function*. Functions in

Lua are *first-class*; that is, a function is treated as just another type, like an integer, or a string.

This idea of a function being simply another type is not particularly unique. JavaScript acts similarly; in fact, aside from using curly braces, the syntax is almost exactly the same. First-class functions stem from the idea of *code as data*. In Lua, a sequence of statements can be treated the same way that a sequence of numbers is treated; that is, as a mutable data type.

## Object-Orientation in Lua

Any introduction to object-orientation usually begins with simple examples. Consider a classical example of a "Person" class in Java:

```java
public class Person {
    public String name;
    public int age;

    public Person(String name, int age){
        this.name = name;
        this.age = age;
    }
}
```

This can be directly converted to Lua as follows:

```lua
Person = {
  name = nil,
  age = 0
}

Person.__index = Person

function Person.new(name, age)
  local self = setmetatable({}, Person)
  self.name = name
  self.age = age
  return self
end
```

The difference is stark. There is no class declaration; instead, a new table is created called `Person`. This table is just another mutable object just any other, but it has special meaning as given by the programmer. This object is a *prototype*; it is the `Person` that all "instances" should base themselves on.

Under the prototype and the following line is a declaration of a constructor (`function Person.new(name, age)...`). Here, `new` is not some kind of keyword, but rather the name of a factory method. Some syntactical sugar is introduced to make Lua look similar to C function declarations; the above segment could be rewritten as:

```
Person = {
  name = nil,
  age = 0,
  new = function(name, age)
    local self = setmetatable({}, Person)
    self.name = name
    self.age = age
    return self
  end
}

Person.__index = Person
```

Inside the constructor, `self` is the new object to be returned from the factory method. Notably, the built-in `setmetatable` method is used on a new table object to initiate the values of the prototype into the new object. The functions however are not copied over, which is where the final line comes in; by setting the `__index` key of the Person prototype to itself, any call to a function of an instance of Person will first check if the function exists within that object, and then, if it does not, it will attempt to reference the function in Person. This reproduces the traditional parent-child hierarchy of object-oriented programming, and prevents unnecessary duplication of data.

With our example, a Person can be trivially instantiated as follows:

```
p1 = Person.new("Bob", 5)
print(p1.name) -- Prints Bob
```

### Inheritance

Consider the Person example in Java from before. Now, consider a “Student” subclass which adds a field for GPA. This may be implemented as follows:

```
public class Student extends Person {
    public float gpa;

    public Student(String name, int age, float gpa){
        super(name, age);
        this.gpa = gpa;
    }
}
```

In Lua, to create a subclass, another prototype should be created, based on the previous prototype:

```lua
Student = Person.new() -- Params will be nil per default
Student.gpa = 0.0

function Student.new(name, age, gpa)
  local self = setmetatable(Person.new(name, age), Student)
  self.gpa = gpa
  return self
end

p2 = Student.new("Jerry", 18, 4.5)
```

In this example, inheritance is not especially useful, but for parent prototypes with many methods connected beyond the constructor, it may be. In addition, methods from the parent can be overridden by redefining the function.

## Conclusion

Lua does not have built-in object-orientation, but instead achieves it through existing constructs. Rather than treating classes as special constructs, prototypes are created as tables. The members of a class are key/value pairs in tables. The first-class nature of prototypes reflects a code as data paradigm; that is, classes, which are represented by tables, are just another type of value in Lua. This contributes towards a consistent feel that makes Lua special to work in.

## References

1. Ierusalimschy, R. (2003). Programming in Lua (1st ed.). Lua.org. Retrieved February 18, 2021, from https://www.lua.org/pil/contents.html
2. Ierusalimschy, R., De Figueiredo, L. H., & Celes, W. (2018). A look at the design of Lua. Communications of the ACM, 61(11), 114-123. doi:10.1145/3186277. Retrieved February 18, 2021, from https://cacm.acm.org/magazines/2018/11/232214-a-look-at-the-design-of-lua/fulltext