

Lua Classes and Object-Oriented Programming Tutorial

Note: This tutorial covers Lua classes, inheritance, and object-oriented programming concepts as used in Fibaro QuickApps and PLua development. Understanding these concepts will help you leverage QuickApp classes effectively and create your own custom classes.

When coding QuickApps on the HC3 or using PLua, we work with **classes** - specifically by extending the `QuickApp` class provided by Fibaro. This tutorial will help you understand how and why classes work the way they do, enabling you to create more sophisticated automation solutions.

Who is this for?

- You're new to Lua and want a gentle intro to classes as used on Fibaro HC3 and PLua
- You prefer lots of examples and practical tips over theory
- You've seen `function Class:method()` and wonder what the colon does

Important: Where does `class` come from?

- Plain Lua does not have a built-in `class` keyword.
- On Fibaro HC3 and in PLua, the environment provides `class` and the QuickApp classes. That's what this tutorial builds on.
- If you run plain Lua elsewhere, you'd create classes with tables and metatables instead.



Table of Contents



Introduction to Classes

- What are Classes?
- Classes vs Instances vs Methods



Building Your First Class

- **Part 1: Basic Class Definition**
 - Creating a Simple Light Class
 - Understanding Method Syntax
 - The Initializer Function
 - Quick Start in PLua / REPL
- **Part 2: Understanding Self and Method Calls**
 - How `:` Syntax Works
 - Behind the Scenes Translation

- Instance Variables
 - Colon vs Dot: Quick Rules
 - Common Errors and Fixes

Advanced Class Features

- **Part 3: Class Inheritance**
 - Creating Subclasses
 - Method Overriding
 - Calling Parent Methods
- **Part 4: QuickApp Class Extension**
 - Extending QuickApp
 - QuickAppChildren
 - Class vs Object Hierarchies





Practical Applications

- **Part 5: Shared Variables and State**
 - Class Variables
 - Shared State Between Instances
- **Part 6: Advanced Class Examples**
 - Global Variable Wrapper Class
 - Property Getters and Setters
 - Operator Overloading

Introduction to Classes

When coding QuickApps on the HC3 or using PLua, we use **classes** - specifically, we extend the `QuickApp` class that Fibaro provides.

Understanding object-oriented programming (OOP) concepts helps you:

-  Leverage the QuickApp class more effectively
-  Create QuickAppChildren with custom behavior
-  Build reusable code abstractions
-  Organize complex automation logic

Key OOP Concepts

Let's define the three fundamental concepts:

- **Class:** A template that tells us how to create objects (like a blueprint)
- **Instance:** An actual object created from a class template (the "Objects" in OOP)
- **Method:** A function that belongs to a class

Think of it this way: If we have a `Dog` class, we can create multiple dog instances - each with their own characteristics but sharing the same methods. For home automation, let's use lights as our example:

```

-- Class definition (the blueprint)
class 'Light'

-- Creating instances (actual light objects)
local livingRoomLight = Light(66) -- Device ID 66
local kitchenLight = Light(77)    -- Device ID 77

```

Each instance has its own data (like device ID) but shares the same methods (like `turnOn`, `turnOff`).

Part 1: Basic Class Definition

Let's create a complete `Light` class that can control switch devices:

```

class 'Light'

-- Constructor/Initializer - called when creating new instances
function Light:__init(deviceId)
    self.id = deviceId
end

-- Methods for controlling the light
function Light:on()
    fibaro.call(self.id, "turnOn")
end

function Light:off()
    fibaro.call(self.id, "turnOff")
end

function Light:toggle()
    -- NOTE: Depending on device type, `fibaro.getValue` may return
    -- like "true"/"false" or numbers. Adjust casting as needed for
    local currentValue = fibaro.getValue(self.id, "value")
    local isOn = (currentValue == true) or (currentValue == "true")
    if isOn then self:off() else self:on() end
end

-- Usage
local light1 = Light(66)
local light2 = Light(77)

light1:on()
light2:off()

```

Understanding the Components

Class Declaration:

```
class 'Light'
```

This creates a new class template named `Light`.

Initializer Method:

```
function Light:__init(deviceId)
    self.id = deviceId
end
```

- `__init` is a special method called when creating instances
- It receives `deviceId` as a parameter and stores it in `self.id`
- `self` is the actual instance we are creating (think table) and we can add keys (variables) to this specific instance. The id, unique for this instance is an example of a specific value we want to store in the instance.
- The double underscore `__` prevents naming conflicts with your own methods, and is the initializer named expected by the Lua/HC3 class implementation

Instance Methods:

```
function Light:on()
    fibaro.call(self.id, "turnOn")
end
```

- `function Light:methodName()` defines a method for the class
- The `:` syntax automatically provides `self` as the first parameter
- `self` refers to the specific instance the method is called on

```
local l = Light(123)
l:on()           -- colon call (recommended)
Light.on(l)      -- dot call with self passed manually
```

Part 2: Understanding Self and Method Calls

The `:` syntax in Lua is syntactic sugar that makes object-oriented programming more natural. Let's see what's happening behind the scenes.

Method Definition Translation

When you write:

```
function Light:on()
    fibaro.call(self.id, "turnOn")
end
```

Lua translates this to:

```
function Light.on(self)
    fibaro.call(self.id, "turnOn")
end
```

Method Call Translation

If you have an instance of Light in a variable named `light1` then when you call:

```
light1:on()
```

Lua translates this to:

```
light1.on(light1)  -- Passes the instance as 'self'
```

Mental Model of Class Construction

Here's a simplified view of how the Light constructor works:

```
-- Conceptual implementation (not actual code)
function Light(deviceId)
    local self = {}          -- Create an empty instance table
    Light.__init(self, deviceId) -- Call initializer and let it pop
    return self              -- Return the new instance
end
```

Instance Variables

Each instance gets its own `self` table with instance variables:

```
local light1 = Light(66)
local light2 = Light(77)

-- Each instance has its own id
print(light1.id)  -- 66
print(light2.id)  -- 77
```

because our initializer creates a new empty table, `self = {}` each time.

: vs . (Colon vs Dot): Quick Rules

- Use a colon when defining methods that operate on an instance:
 - Definition: `function ClassName:method(a, b)`
 - Call: `obj:method(a, b)`
- Under the hood, `obj:method(a)` becomes `ClassName.method(obj, a)`.
- If you use a dot in the definition, you must pass `self` manually:
 - Definition: `function ClassName.method(self, a)`
 - Call: `ClassName.method(obj, a)`

In short: Prefer the colon for both defining and calling instance methods—it's less error-prone.

Common Errors and Fixes

- Error: attempt to index a nil value (self)
 - Cause: You defined the method with `.` but called with `:` (or vice versa), so `self` is nil.
 - Fix: Use `function Class:method(...)` and call with `obj:method(...)`.
- Error: attempt to call a nil value (method 'on')
 - Cause: The instance doesn't have that method in its class (typo or not in scope).
 - Fix: Check the method name and that the class was defined before creating the instance.
- Error: Device values don't toggle correctly
 - Cause: `fibaro.getValue` may return strings; you're comparing with booleans or numbers.
 - Fix: Normalize the value (see `toggle` above) or store your own `self.isOn` state.

Part 3: Class Inheritance

Inheritance allows you to create specialized classes based on existing ones.

Let's create a `DimLight` class that extends our `Light` class:

```
class 'DimLight'(Light)  -- DimLight inherits from Light

function DimLight:__init(deviceId)
    Light.__init(self, deviceId)  -- Call parent initializer
end

-- New method specific to dimmable lights
function DimLight:dim(value)
    fibaro.call(self.id, "setValue", value)
end

-- Override parent method with additional functionality
function DimLight:on()
    Light.on(self)  -- Call parent method
    self:debug("Dimmable light turned on")
end

-- Usage
local dimmer = DimLight(109)
dimmer:on()      -- Inherited method (with override)
dimmer:dim(50)   -- New method
dimmer:off()     -- Inherited method
```

Important to call the parent's initializer method in your own initializer. Like `DimLight` calls `Light`'s initializer in the example above.

Key Inheritance Concepts

- **Parent Class:** The class being inherited from (`Light`)
- **Child Class:** The class doing the inheriting (`DimLight`)
- **Override:** Redefining a parent method in the child class
- **Super Call:** Calling the parent's version of a method using `Parent.method(self, ...)`

Creating Class Hierarchies

You can build complex inheritance trees:

```
-- Base class
class 'Device'

-- First level inheritance
class 'Light'(Device)
class 'Sensor'(Device)

-- Second level inheritance
class 'DimLight'(Light)
class 'ColorLight'(Light)
class 'MotionSensor'(Sensor)
```

How field and method lookup works (under the hood)

When you access `obj.someField` :

- Lua looks in the instance table first (things you put in `self.xxx`).
- If not found, it looks in the class's methods table via `__index` .
- If still not found, it walks up to the parent class (and so on).

That's why class variables like `Light.version` live on the class, and instance variables like `self.id` live on the object.

Part 4: QuickApp Class Extension

In QuickApp development, you're extending Fibaro's `QuickApp` class:

```
--%%name: MySmartSwitch
--%%type: com.fibaro.binarySwitch

-- This extends the QuickApp class
function QuickApp:OnInit()
    self:debug("MySmartSwitch initializing...")

    -- Initialize your custom properties
    self.lightDevice = 66
    self.motionSensor = 77
```

```

    self:setupEventHandlers()
end

function QuickApp:setupEventHandlers()
    -- Custom method for organization
    self:debug("Setting up event handlers")
end

function QuickApp:turnOn()
    -- Override default turnOn behavior
    self:updateProperty("value", true)
    fibaro.call(self.lightDevice, "turnOn")
end

```

We say it `extends` as it adds methods to an existing class, QuickApp in this case. We don't `inherit` from QuickApp.

QuickApp Startup Process

When a QuickApp starts, Fibaro essentially does:

```

function startQuickApp(deviceId)
    local device = api.get("/devices/" .. deviceId)
    local qa = QuickApp(device)  -- Create QuickApp instance

    if qa.onInit then
        qa:onInit()  -- Call your onInit method
    end

    quickApp = qa  -- Make globally available
end

```

QuickAppChildren

QuickAppChildren allow you to create child devices:

```

class 'HueSwitch'(QuickAppChild)

function HueSwitch:__init(hueId, device)
    QuickAppChild.__init(self, device)
    self.hueId = hueId
end

function HueSwitch:turnOn()
    HueFunctions.turnOn(self.hueId)
    self:updateProperty("value", true)
end

function HueSwitch:turnOff()
    HueFunctions.turnOff(self.hueId)
end

```



```
self:updateProperty("value", false)
end
```



Class vs Object Hierarchies

Class Hierarchy - How classes inherit from each other:

```
QuickAppBase
├─ QuickApp
├─ QuickAppChild
│   └─ HueSwitch
```

Object Hierarchy - How instances relate at runtime:

- QuickApp instance has `self.childDevices` table
- Each child has `self.parent` reference back to QuickApp

```
-- In your QuickApp
for id, child in pairs(self.childDevices) do
    self:debug("Child:", id, "Name:", child.name)
end

-- In a QuickAppChild
self.parent:setVariable("status", "active")
```

Part 5: Shared Variables and State

You can create variables shared between all instances of a class:

```
class 'Light'

-- Class variable - shared by all instances
Light.version = "1.0"
Light.shared = { debug = true, count = 0 }

function Light:__init(deviceId)
    self.id = deviceId
    Light.shared.count = Light.shared.count + 1 -- Track instance count
end

function Light:on()
    if Light.shared.debug then
        print("Light " .. self.id .. " turned on")
    end
    fibaro.call(self.id, "turnOn")
end

-- Usage
local light1 = Light(66)
local light2 = Light(77)
```

```

print(light1.version)      -- "1.0"
print(Light.shared.count)  -- 2

-- Change shared state
Light.shared.debug = false
light1:on()  -- No debug output

```

! Shared vs Instance Variables

```

-- Instance variable - each instance gets its own copy
light1.brightness = 50
light2.brightness = 75

-- Shared variable - all instances see the same value
Light.shared.totalLights = 10

```

Part 6: Advanced Class Examples

Global Variable Wrapper Class

Let's create a class that makes working with Fibaro global variables easier:

```

class 'Global'

function Global:__init(name, create)
    self.name = name

    -- Create global if it doesn't exist
    if not api.get("/globalVariables/" .. name) and create then
        api.post("/globalVariables", {name = name, value = ""})
    end

    -- Property with getter and setter
    self.value = property(
        function(self) -- getter
            local value = fibaro.getGlobalVariable(self.name)
            return self:castValue(value)
        end,
        function(self, value) -- setter
            fibaro.setGlobalVariable(self.name, json.encode(value))
        end
    )
end

function Global:castValue(str)
    local constants = {'true' = true, 'false' = false}

    if str == nil then
        return nil
    elseif constants[str] then

```

```

        return constants[str]
    elseif str:match("^[%{%[]}") then
        return json.decode(str) -- JSON table
    else
        return tonumber(str) or str -- Number or string
    end
end

function Global:__tostring()
    return "Global:" .. self.name
end

-- Usage
local config = Global('myConfig', true)
config.value = {lights = {66, 77}, timeout = 300}
print(config.value.lights[1]) -- 66

```

Property Getters and Setters

The `property()` function creates special instance variables:

```

self.value = property(
    function(self) return fibaro.getGlobalVariable(self.name) end,
    function(self, value) fibaro.setGlobalVariable(self.name, value)
)

```

Now `config.value = "test"` calls the setter, and `print(config.value)` calls the getter.

Operator Overloading

You can define how operators work with your classes using special methods:

```

class 'Date'

function Date:__init(timeString)
    self.time = self:parseDate(timeString) or os.time()
end

function Date:__tostring()
    return os.date("%c", self.time)
end

function Date:__eq(other) -- Equality operator ==
    return self.time == other.time
end

function Date:__lt(other) -- Less than operator <
    return self.time < other.time
end

-- Usage

```

```
local date1 = Date("2023/12/25 12:00")
local date2 = Date("2023/12/25 13:00")

print(date1 == date2)  -- false
print(date1 < date2)   -- true
```

Available Operator Overrides

- `__add`, `__sub`, `__mul`, `__div` - Arithmetic operators
- `__eq`, `__lt`, `__le` - Comparison operators
- `__tostring` - String conversion
- `__call` - Make instances callable like functions
- `__len` - Length operator `#`

Best Practices

Do's

1. Use descriptive class names: `MotionSensorController` instead of `MSC`
2. Initialize all instance variables in `__init` :

```
function Light:__init(deviceId)
    self.id = deviceId
    self.brightness = 0
    self.isOn = false
end
```

3. Use inheritance for "is-a" relationships:

- `DimLight` is a `Light` 
- `Room` has `Lights` (composition, not inheritance)

4. Call parent initializers in child classes:

```
function DimLight:__init(deviceId)
    Light.__init(self, deviceId)  -- Always call parent
    self.dimLevel = 0
end
```

5. Use `self` consistently: Always access instance variables via `self.variable`

Don'ts

1. Don't forget the colon in method definitions: Use `function Class:method()` not `function Class.method()`
2. Don't access class variables directly on instances: Use `Class.variable` instead of `instance.variable` for shared state
3. Don't create deep inheritance hierarchies: Keep it simple, usually 2-3 levels max

4. Don't mix class and instance variables carelessly:

```
-- Bad: Confusing mix
Light.count = 0 -- Class variable
function Light:__init(id)
    self.count = Light.count + 1 -- Creates instance variable!
end

-- Good: Clear separation
Light.shared = { totalCount = 0 }
function Light:__init(id)
    Light.shared.totalCount = Light.shared.totalCount + 1
end
```

Practice Exercises

Try a few small, confidence-building tasks:

1. Add a `blink(times, delayMs)` method to `Light` that toggles on/off repeatedly using `setTimeout`.
2. Create a `Room` class that holds a list of `Light` instances and has `onAll()` / `offAll()` methods.
3. Make a `DimLight` override of `on()` that also sets brightness to 50 by default.
4. Add a class variable `Light.shared.debug` that controls whether methods print what they're doing.
5. Convert one colon-defined method to dot form and back, to see how passing `self` works.

Tip: Use interactive REPL mode to iterate quickly.

Summary

Understanding Lua classes enables you to:

- **Extend QuickApp effectively** with organized, maintainable code
- **Create QuickAppChildren** with custom behavior for complex devices
- **Build reusable abstractions** for common HC3 operations
- **Organize complex automation logic** using object-oriented principles

Quick Reference

```
-- Define class
class 'ClassName'

-- Inheritance
class 'ChildClass'(ParentClass)

-- Initializer
```

```

function ClassName:__init(params)
    ParentClass.__init(self)      -- If parent, call parent's in
    -- Initialize instance variables
    self.variable = value
end

-- Methods
function ClassName:methodName(params)
    -- Method body
end

-- Create instance
local obj = ClassName(params)

-- Call methods
obj:methodName()

```

Classes are powerful tools for creating clean, maintainable QuickApps. Start with simple examples and gradually build more complex class hierarchies as your automation needs grow! 🚀

? FAQ

- Where does `class` come from?
 - It's provided by the Fibaro HC3 runtime and PLua environment. Plain Lua doesn't have a built-in `class` keyword. There are different class model implementations for Lua, but the one described here is what we get with HC3.
- Do I always need to call the parent initializer?
 - Yes, if your class inherits from another. Call it first:


```
Parent.__init(self, ...)
```
- Should I use `:` or `.`?
 - For instance methods, prefer `:` in both definitions and calls. Use `.` only if you know you need to pass `self` explicitly.
- What if my device values don't behave like booleans?
 - Normalize strings/numbers to booleans, or track your own `self.isOn` state to simplify logic.
- Can I use these classes outside of QuickApps?
 - On the HC3 you can only use it in QuickApps. Scenes don't have class support. Yes, in PLua or any environment that provides the same `class` facility. For plain Lua, you would implement similar behavior with metatables.