# The Anatomy of QuickApps - Part 1

> Note: This series explores QuickApp internals. It focuses not only on "call this function to do this," but on why calling a function produces a particular result.

We'll gradually dive into QuickApps and understand how they work under the hood. This first part is mostly an introduction to Lua functions and objects—the foundation we need. The next part goes deeper into the QuickApp class and how it works. It will get progressively more challenging! 😊

## 📖 Table of Contents

---

# Introduction

**QuickApps are written in the programming language Lua.** Let's look behind the curtain of a QuickApp and understand how it works.

## Prerequisites

To follow along, you should have:

- Basic understanding of Lua
- Experience making a simple QuickApp
- Knowledge of Lua tables (helpful but we'll explain most concepts)

Even for seasoned Lua coders, this may explain why Lua behaves the way it does sometimes. Corrections and feedback are welcome!

## What We'll Cover

1. **Lua functions recap**
2. **Lua tables and how they are constructed/accessed**

3. **Object-oriented programming in Lua**
4. **Classes and objects**
5. **Finally - the QuickApp class and how it works internally**

## Part 1: Lua Functions Recap

First, QuickApps are based on a Lua object-oriented model using “classes.” But what is a class in Lua? To answer that, we need to take a short tour through Lua fundamentals.

### Basic Function Definition

We know that:

```
function test(x)
    return x + x
end
```

This defines a Lua function named `test` that takes one argument `x` and returns `2*x`.

> **Note**: `x` has to be a number, or we get an error. Only numbers can be added.

We can call the function:

```
x = test(21)
print(x)  -- prints 42
```

Don't be confused that the parameter of `test` is `x` and we assign `x` the result. It's a different `x`. We say that the parameter `x` is **scoped** to the function `test` (more on this later).

We can create locally scoped variables inside `test` also:

```
function test(x)
    local y = 8
    return y + x
end
```

### ⚠️ Important: Function Definition Order

When Lua runs your code, it runs **top to bottom**, starting with the first line. This means functions must be defined before you can call them.

```
-- ❌ This will NOT work:
x = test(21)
function test(x) return x + x end
```

Lua will complain that `test` doesn't exist when it sees `x = test(21)`. The function is only defined on the next line.

# Part 2: Functions as First-Class Values

## Anonymous Functions

What happens when a function is defined? It turns out that:

```
function test(x) return x + x end
```

is just a convenient way that Lua allows us to write:

```
test = function(x) return x + x end
```

From this we learn two important things:

1. **We have "anonymous" functions** (or just "functions" for short)
2. **A "named" function is just a variable assigned an anonymous function**

## Functions are Data Types

A function in Lua is a data type just like any other. In fact, Lua has only eight basic types:

| Type | Description | Example |
|---|---|---|
| `boolean` | Logical true/false | `print(type(true))` → `"boolean"` |
| `string` | Text data | `print(type("Hello"))` → `"string"` |
| `number` | Numeric values | `print(type(42))` → `"number"` |
| `table` | Arrays and key-value maps | `print(type({}))` → `"table"` |
| `function` | Function objects | `print(type(function() end))` → `"function"` |
| `userdata` | C/C++ data | `print(type(QuickApp))` → `"userdata"` |
| `thread` | Coroutines (not allowed on HC3) | - |
| `nil` | "Nothing" value | `print(type(nil))` → `"nil"` |

> HC3 note: The `class` function creates objects of type `userdata`—these are defined in C/C++ code.

## Passing Functions as Arguments

Since functions are values, we can pass them as arguments:

```
function test(f)
    return 11 + f(21)
```

```lua
end

x = test(function(x) return x + x end)
print(x)  -- prints 53 (11 + 21 + 21)
```

What's happening:

1. We define `test` that takes parameter `f` (assumed to be a function)
2. `test` calls `f(21)` and adds 11 to the result
3. We call `test` with an anonymous function that doubles its argument
4. Result: `11 + (21 + 21) = 53`

Alternative approach:

```lua
function test(f)
    return 11 + f(21)
end

function myFun(x)
    return x + x
end

x = test(myFun)  -- Same result: 53
```

---

# Part 3: Variable Arguments and Multiple Returns

## Variable Arguments with `...`

Normally we define functions with fixed arguments:

```lua
function sum(x, y)
    return x + y
end
```

But what if we want to sum any number of arguments? We could pass an array:

```lua
function sum(args)
    local s = 0
    for _, x in ipairs(args) do
        s = s + x
    end
    return s
end

sum({1, 2, 3, 4, 5})  -- or sum{1, 2, 3, 4, 5}
```

> **Lua Tip**: You can omit parentheses when passing a single table to a function.

Even better—use variable arguments:

```lua
function sum(...)
    local args = {...}  -- Convert ... to table
    local s = 0
    for _, x in ipairs(args) do
        s = s + x
    end
    return s
end

sum(1, 2, 3, 4, 5)  -- Much cleaner!
```

### Named Parameters Pattern

This style is common for “named parameters”:

```lua
function test(args)
    print(string.format("My name is %s and I'm a %s",
                        args.name, args.animal))
end

test{name="Bob", animal="dog"}
-- Prints: "My name is Bob and I'm a dog"
```

### Multiple Return Values

Functions can return multiple values:

```lua
function test()
    return 42, 17
end

a, b = test()  -- a=42, b=17

-- Functions can use multiple returns as arguments:
print(sum(test()))  -- sum receives 42 and 17, prints 59
```

---

## Part 4: Global vs Local Variables

### Global Variables

Global variables are stored in Lua's global context table `_G`:

```lua
test = 42
-- Equivalent to: _G["test"] = 42

function test(x) return x + x end
-- Equivalent to: _G["test"] = function(x) return x + x end
```

**Direct access to global table:**

```lua
_G["test"] = function(x) return x + x end
print(test(21))  -- prints 42
```

### Local Variables and Scope

```lua
local test1 = 42           -- Local variable (not in _G)
function test2(x)          -- Global function (in _G)
    return x + x
end
```

Local scopes are created by:

- `do ... end` blocks
- `if then ... else ... end` statements
- `repeat ... until` loops
- `while ... do ... end` loops
- Function bodies

**Example of variable shadowing:**

```lua
x = 6
do
    local x = 3
    print(x)  -- prints 3 (local shadows global)
end
print(x)      -- prints 6 (global x unchanged)
```

---

## Part 5: Forward Declarations and Mutual Recursion

### The Problem

Consider two functions that call each other:

```lua
-- ✅ This works (both global):
function foo(x)
    if x == 0 then
        bar(x)
    else
        print(x)
    end
end

function bar(x)
    if x == 0 then
        foo(42)
    end
end

foo(0)  -- prints 42
```

**But with local functions:**

```lua
-- ❌ This crashes:
do
    local function foo(x)
        if x == 0 then bar(x) else print(x) end
    end

    local function bar(x)
        if x == 0 then foo(42) end
    end

    foo(0)  -- Error: "attempt to call global 'bar' (a nil value)"
end
```

Why it fails: When `foo` is defined, `bar` doesn't exist yet as a local variable, so `bar` is assumed to be global (but it doesn't exist there either).

### The Solution: Forward Declaration

```lua
-- ✅ This works:
do
    local bar  -- Forward declare

    local function foo(x)
        if x == 0 then bar(x) else print(x) end
    end

    function bar(x)  -- Note: no 'local' keyword here
        if x == 0 then foo(42) end
    end

    foo(0)  -- Works! prints 42
end
```

**Equivalent to:**

```lua
do
    local bar
    local foo = function(x)
        if x == 0 then bar(x) else print(x) end
    end
    bar = function(x)
        if x == 0 then foo(42) end
    end
    foo(0)
end
```

## Part 6: Function Redefinition and Closures

## Redefining Built-in Functions

Since named functions are just variables with function values, we can redefine built-ins:

```
do
    local oldTostring = tostring  -- Save original
    function tostring(value)
        return "V:" .. oldTostring(value)
    end
end

print(42)  -- prints "V:42"
```

Practical example—better table printing:

```
do
    local oldTostring = tostring
    function tostring(value)
        if type(value) == 'table' then
            return json.encode(value)
        else
            return oldTostring(value)
        end
    end
end

print({a=8})  -- prints '{"a":8}' instead of "table: 0x7676876"
```

## Protecting Against Redefinition

To protect your functions from redefinition:

```
do
    local myTostring = tostring  -- Capture current definition
    function myFun(x)
        return "[" .. myTostring(x) .. "]"
    end
end
-- Even if someone redefines tostring later, myFun still works
```

## Closures

Functions can “close over” local variables:

```
x = 42
function foo(y) return x + y end
print(foo(8))  -- prints 50

do
    local x = 42
    function foo(y) return x + y end  -- Captures local x
```

```lua
end
x = 55  -- Global x
print(foo(8))  -- Still prints 50! Uses captured local x
```

**Shared closures:**

```lua
do
    local x = 42
    function getXplusY(y) return x + y end
    function setX(y) x = y end
end

print(getXplusY(8))  -- prints 50
setX(10)
print(getXplusY(8))  -- prints 18
```

Both functions share the same local variable `x`—it's like a private shared variable.

**Function factories:**

```lua
function makeAdder(x)
    return function(y) return x + y end
end

add42 = makeAdder(42)
print(add42(8))  -- prints 50
```

The returned function "remembers" the value of `x` (42) from when it was created. This combination of a function plus captured variables is called a **closure**.

---

## Summary: Functions in Lua

If you've followed along this far, you have a solid grasp of functions in Lua:

✅ **Functions are first-class values** - can be assigned, passed, and returned
✅ **Global vs local scope** and variable shadowing
✅ **Forward declarations** for mutual recursion
✅ **Closures** capture local variables
✅ **Variable arguments** with `...`
✅ **Multiple return values**

**Next up:** We'll use these concepts to build object-oriented programming in Lua, which will lead us to understanding QuickApps!

---

## Part 7: Object-Oriented Programming

Now let's do some object-oriented programming! Lua doesn't have OO built in, but it provides the tools to build your own OO model.

What is an object? An object is an encapsulation of data with associated functions (methods).

## Tables as Data Containers

We can make a Lua table:

```
test = {['a'] = 42, ['y'] = 17}
```

The variable `test` is assigned a key-value table where:

- Key `'a'` → value `42`
- Key `'y'` → value `17`

Important: The table is constructed when this statement runs:

```
function foo(x) return x + 4 end
test2 = {['a'..'b'] = foo(66)}
-- Creates: {['ab'] = 70}
```

## Table Access Syntax

Bracket notation (always works):

```
print(test['a'])  -- prints 42
print(test['y'])  -- prints 17
```

Dot notation (when the key is a valid identifier):

```
print(test.a)  -- prints 42
print(test.y)  -- prints 17
```

When dot notation doesn't work:

```
test = {['a b'] = 42, ['dörr'] = 99}
-- ❌ Can't do: test.'a b' or test.dörr
-- ✅ Must use: test['a b'] and test['dörr']
```

## Table Construction Shortcuts

String keys:

```
test = {a = 42, y = 17}  -- Same as {['a'] = 42, ['y'] = 17}
```

Adding or modifying keys:

```
test.a = 18  -- Modify existing
test.c = 77  -- Create new
```

Nested tables:

```
test = { a = { b = 8 } }
print(test.a.b)  -- prints 8
```

### Arrays vs Hash Tables

Arrays (consecutive numeric keys):

```
test = {[1]="a", [2]="b", [3]="c"}
-- Shortcut: test = {"a", "b", "c"}

print(test[2])    -- prints "b"
-- Note: test.2 is invalid syntax
```

> Lua arrays start at index 1, not 0!

## Part 8: Objects with Methods

### Basic Object Pattern

```
test1 = {a = 42, y = 17}
test2 = {a = 43, y = 18}

print(test1.a)  -- prints 42
print(test2.a)  -- prints 43
```

We have two objects encapsulating their own data.

### Adding Functions to Objects

```
test1 = {
    a = 42,
    y = 17,
    f = function() return test1.a + test1.y end
}
test2 = {
    a = 43,
    y = 18,
    f = function() return test2.a + test2.y end
}

print(test1.f())  -- prints 59
print(test2.f())  -- prints 61
```

Problem: Each function must reference its specific table ( `test1` , `test2` ). The functions are nearly identical!

### Shared Functions with Parameters

```
local f = function(tab) return tab.a + tab.y end

test1 = {a = 42, y = 17, f = f}
test2 = {a = 43, y = 18, f = f}

print(test1.f(test1))  -- prints 59
print(test2.f(test2))  -- prints 61
```

Better! Now we share one function, but we must pass the table as an argument.

### The `:` Syntax Sugar

Lua provides special syntax for this common pattern:

```
test1:f()  -- Same as: test1.f(test1)
```

With the `:` syntax:

```
local f = function(self) return self.a + self.y end

test1 = {a = 42, y = 17, f = f}
test2 = {a = 43, y = 18, f = f}

print(test1:f())  -- prints 59
print(test2:f())  -- prints 61
```

The first parameter is automatically the table itself, conventionally named `self`.

### Function Definition with `:`

Defining methods directly:

```
test1 = {a = 42, y = 17}

function test1:f()
    return self.a + self.y
end

print(test1:f())  -- prints 59
```

This is equivalent to:

```
test1.f = function(self)
    return self.a + self.y
end
```

With additional parameters:

```
function test1:f(x)
    return self.a + self.y + x
end
-- Same as: test1.f = function(self, x) return self.a + self.y + x e
```

## Part 9: Classes and Object Creation

We want a function that creates objects of a specific type—this is called a class.

### Simple Class Implementation

```lua
function createClass(template)
    return template
end

function createObject(class, initValues)
    local obj = {}  -- Create new table

    -- Copy class template
    for key, value in pairs(class) do
        obj[key] = value
    end

    -- Override with initial values
    for key, value in pairs(initValues) do
        obj[key] = value
    end

    return obj
end
```

### Using Our Class System

```lua
-- Create a class
myClass = createClass({a = 42, b = 17})

-- Add methods to the class
function myClass:test()
    print(self.a + self.b)
end

-- Create objects
obj1 = createObject(myClass, {a = 10})
obj2 = createObject(myClass, {a = 11})

obj1:test()  -- prints 27 (10 + 17)
obj2:test()  -- prints 28 (11 + 17)
```

**What happens:**

1. `myClass` template: `{a = 42, b = 17, test = function(self)...}`
2. `obj1` gets: `{a = 10, b = 17, test = function(self)...}`
3. `obj2` gets: `{a = 11, b = 17, test = function(self)...}`

---

## Part 10: QuickApps - Finally!

Now we can understand how QuickApps work. When the HC3 starts your QuickApp:

### Conceptual QuickApp lifecycle

```lua
-- 1. HC3 creates the QuickApp class (simplified)
QuickApp = createClass({...})  -- HC3 does this

-- 2. HC3 adds built-in methods
function QuickApp:debug(...)
    -- Built-in debug implementation
end
-- ... more built-in functions

--- 3. Your code runs ---

-- Your methods get added to the QuickApp class
function QuickApp:turnOn()
    -- Your implementation
end

function QuickApp:onInit()
    self:debug("Started")
end

--- End of your code ---

-- 4. HC3 creates your QuickApp object
local quickApp = createObject(QuickApp, {id = deviceId})

-- 5. HC3 calls your initialization
if quickApp.onInit then
    quickApp:onInit()
end
```

## Real QuickApp implementation

In reality, HC3 uses a built-in `class` function instead of our simple version:

```lua
class "QuickApp"(QuickAppBase)  -- Inherit from QuickAppBase

function QuickApp:debug(...)
    -- Built-in methods
end

--- Your code ---

function QuickApp:turnOn()
    -- Your methods
end

function QuickApp:onInit()
    self:debug("Started")
    -- self.id is automatically set to your device ID
```

```lua
end

--- End of your code ---

-- HC3 creates and initializes your QuickApp
local quickApp = QuickApp()  -- Create object
if quickApp.onInit then
    quickApp:onInit()  -- Call your onInit
end
```

### Key differences from our implementation

1. **Built-in `class` function** creates objects of type `"userdata"` (protected)
2. **Can't introspect** - you can't loop over keys or convert to JSON
3. **Inheritance support** - QuickApp inherits from QuickAppBase
4. **Automatic properties** - `self.id` is set to your device ID

### Accessing QuickApp properties

```lua
function QuickApp:onInit()
    self:debug(self.id)    -- Your device ID
    self:debug(self.name)  -- Your device name
    -- But you can't do: print(json.encode(self))
end
```

---

## Summary: Understanding QuickApps

You now understand:
✅ **Tables** - Lua's flexible data structure
✅ **Object-oriented programming** in Lua using tables and functions
✅ **The `:` syntax** for method calls with implicit `self`
✅ **Classes and objects** - templates and instances
✅ **QuickApp structure** - it's a class with your methods added
✅ **Object lifecycle** - class definition → method addition → object creation → initialization
QuickApps are just Lua objects. When you define `function QuickApp:onInit()`, you're adding a method to the QuickApp class. When HC3 runs your code, it creates an instance of that class and calls your methods.

---

## What's Next?

In Part 2, we'll dive deeper into:

- **Inheritance** and how QuickAppBase works
- **QuickAppChildren** implementation
- **Advanced QuickApp patterns**

- **Method overriding** and calling parent methods

The foundation you've built here will make everything else much clearer! 🎉

# 📚 Quick Reference Index

## Key Concepts

| Concept | Section | Description |
|---|---|---|
| Anonymous Functions | Part 2 | `function(x) return x + x end` |
| Named Functions | Part 1 | `function test(x) return x + x end` |
| Variable Arguments | Part 3 | `function sum(...) local args = {...} end` |
| Global Variables | Part 4 | Stored in `_G` table |
| Local Variables | Part 4 | Scoped to blocks/functions |
| Forward Declaration | Part 5 | `local bar` before defining `bar` |
| Closures | Part 6 | Functions that capture local variables |

## Lua Syntax

| Syntax | Section | Example |
|---|---|---|
| Table Access | Part 7 | `table.key` or `table['key']` |
| Method Call | Part 8 | `object:method()` ≡ `object.method(object)` |
| Method Definition | Part 8 | `function object:method() ... end` |
| Table Construction | Part 7 | `{a = 1, b = 2}` or `{"item1", "item2"}` |

## Data Types

| Type | Section | Description | Example |
|---|---|---|---|
| `boolean` | Part 2 | true/false | `true`, `false` |
| `string` | Part 2 | Text data | `"Hello"` |
| `number` | Part 2 | Numeric values | `42`, `3.14` |
| `table` | Part 7 | Arrays and objects | `{a = 1}`, `{"x", "y"}` |

| | | | |
|---|---|---|---|
| `function` | Part 2 | Function objects | `function() end` |
| `userdata` | Part 10 | C/C++ objects | `QuickApp` |
| `nil` | Part 2 | Nothing/absence | `nil` |

## QuickApp Patterns

| Pattern | Section | Usage |
|---|---|---|
| **Method Definition** | Part 10 | `function QuickApp:onInit() ... end` |
| **Accessing Properties** | Part 10 | `self.id`, `self.name` |
| **Calling Methods** | Part 10 | `self:debug("message")` |
| **Object Creation** | Part 10 | `local qa = QuickApp()` |