

# Lua Language Server | Wiki

## Annotations

Add additional context to your code and type checking.

The language server does its best to infer types through contextual analysis, however, sometimes manual documentation is necessary to improve completion and signature information. It does this through LuaCATS (Lua Comment And Type System) annotations, which are based off of [EmmyLua annotations](#).

Annotations are prefixed with `---`, like a Lua comment with one extra dash.

## Annotation Formatting

Annotations support most of the [markdown syntax](#). More specifically, you can use:

- headings
- bold text
- italic text
- struckthrough text
- ordered list
- unordered list
- blockquote
- code
- code block
- horizontal rule
- link
- image

There are many ways to add newlines to your annotations. The most bulletproof way is to simply add an extra line of just `---`, although this functions like a paragraph break, not a newline.

The below methods can be added to the end of a line:

- HTML `<br>` tag (recommended)
- `\n` newline escape character
- Two trailing spaces (may be removed by formatting tools)
- Markdown backslash `\` ( [not recommended](#) )

## Referring to symbols

As of [v3.9.2](#) you can refer to symbols from your workspace in markdown descriptions using markdown links. Hovering the described value will show a hyperlink that, when clicked, will take you to where the symbol is defined.

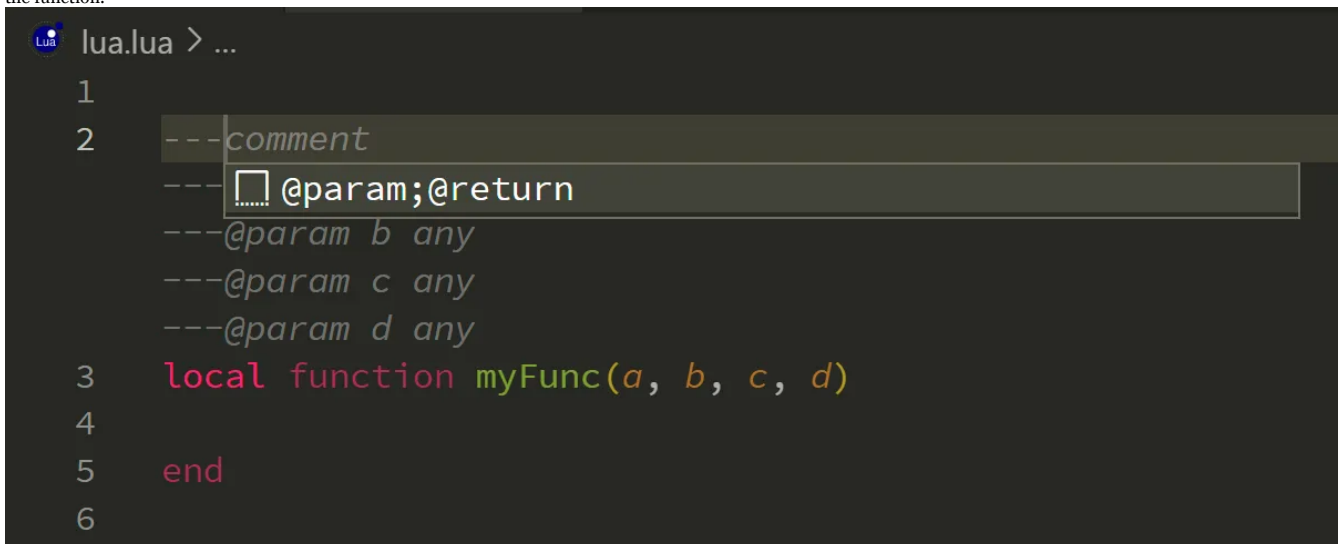
lua

```
---@alias MyCustomType integer
```

```
---Calculate a value using [my custom type](lua://MyCustomType)
function calculate(x) end
```

## Tips

- If you type `---` one line above a function, you will receive a suggested snippet that includes [@param](#) and [@return](#) annotations for each parameter and return value found in the function.



The screenshot shows a code editor with a Lua file named 'lua.lua'. The cursor is positioned at the end of a line starting with '---'. A hover tooltip is displayed, showing a list of suggested annotations: '@param', '@return', '@param b any', '@param c any', and '@param d any'. Below the tooltip, the code snippet is visible: 'local function myFunc(a, b, c, d)' followed by 'end' on the next line. The editor has a dark theme and a sidebar on the left showing the file 'lua.lua'.

## Documenting Types

Properly documenting types with the language server is very important and where a lot of the features and advantages are. Below is a list of all recognized Lua types (regardless of [version in use](#)):

- nil
- any
- boolean
- string
- number
- integer

- function
- table
- thread
- userdata
- lightuserdata

You can also simulate [classes](#) and [fields](#) and even [create your own types](#).

Adding a question mark (?) after a type like `boolean?` or `number?` is the same as saying `boolean|nil` or `number|nil`. This can be used to specify that something is either a specified type **or** `nil`. This can be very useful for function returns where a value **or** `nil` can be returned.

Below is a list of how you can document more advanced types:

Type	Document As:
Union Type	<code>TYPE_1   TYPE_2</code>
Array	<code>VALUE_TYPE[]</code>
Tuple	<code>[VALUE_TYPE, VALUE_TYPE]</code>
Dictionary	<code>{ [string]: VALUE_TYPE }</code>
Key-Value Table	<code>table&lt;KEY_TYPE, VALUE_TYPE&gt;</code>
Table literal	<code>{ key1: VALUE_TYPE, key2: VALUE_TYPE }</code>
Function	<code>fun(PARAM: TYPE): RETURN_TYPE</code>

Unions may need to be placed in parentheses in certain situations, such as when defining an array that contains multiple value types:

```
Lua
---@type (string | integer)[]
local myArray = {}
```

## Understanding This Page

To get an understanding of how to use the annotations described on this page, you'll need to know how to read the Syntax sections of each annotation.

Symbol	Meaning
<code>&lt;value_name&gt;</code>	A required value that you provide
<code>[value_name]</code>	Everything inside is optional
<code>{value_name...}</code>	This value is repeatable
<code>value_name   value_name</code>	The left <b>or</b> right side are valid

Any other symbols are syntactically required and should be copied verbatim.

If this is confusing, take a look at a couple examples under an annotation and it should make more sense.

## Annotations List

Below is a list of all annotations recognized by the language server:

### @alias

An alias can be used to create your own type. You can also use it to create an enum that does not exist at runtime. For an enum that *does* exist at runtime, see [@enum](#).

#### Syntax

```
Lua
---@alias <name>
---| '<value>' [# description]
```

#### Examples

##### ▼ Simple alias

```
Lua
---@alias userID integer The ID of a user
```

##### ▼ Custom Type

```
Lua
---@alias modes "r" | "w"
```

##### ▼ Custom Type with Descriptions

```
Lua
---@alias DeviceSide
---| "'left'" # The left side of the device
---| "'right'" # The right side of the device
---| "'top'" # The top side of the device
---| "'bottom'" # The bottom side of the device
---| "'front'" # The front side of the device
---| "'back'" # The back side of the device

---@param side DeviceSide
local function checkSide(side) end
```

##### ▼ Literal Custom Type

```
Lua
local A = "Hello"
local B = "World"

---@alias myLiteralAlias `A` | `B`

---@param x myLiteralAlias
function foo(x) end
```

##### ▼ Literal Custom Type with Descriptions

```

Lua

local A = "Hello"
local B = "World"

---@alias myLiteralAlias `A` | `B`

---@param x myLiteralAlias
function foo(x) end

```

## @as

Force a type onto an expression.

### Syntax

```
--[[@as <type>]]
```

### Examples

#### ▼ Override Type

```

Lua

---@param key string Must be a string
local function doSomething(key) end

local x = nil

doSomething(x --[[@as string]])

```

## @async

Mark a function as being asynchronous. When [hint.await](#) is true, functions marked with @async will have an `await` hint displayed next to them when they are called. Used by diagnostics from the [await group](#).

### Syntax

```
---@async
```

### Examples

#### ▼ Mark Function Async

```

Lua

---@async
---Perform an asynchronous HTTP GET request
function http.get(url) end

```

## @cast

Cast a variable to a different type or types.

### Syntax

```
---@cast <value_name> [+|-]<type|?>[, [+|-]<type|?>...]
```

### Examples

#### ▼ Simple Cast

```

Lua

---@type integer | string
local x

---@cast x string
print(x) --> x: string

```

#### ▼ Add Type

```

Lua

---@type integer
local x

---@cast x +boolean
print(x) --> x: integer | boolean

```

#### ▼ Remove Type

```

Lua

---@type integer|string
local x

---@cast x -integer
print(x) --> x: string

```

#### ▼ Cast Multiple Types

```

Lua

---@type string
local x --> x: string

---@cast x +boolean, +number
print(x) --> x:string | boolean | number

```

#### ▼ Cast Possibly nil

```

Lua

---@type string
local x

---@cast x +?
print(x) --> x:string?

```

## @class

Define a class. Can be used with [@field](#) to define a table structure. Once a class is defined, it can be used as a type for [parameters](#), [returns](#), and more. A class can also inherit one or more parent classes. Marking the class as `(exact)` means fields cannot be injected after the definition.

### Syntax

```
---@class [(exact)] <name>[: <parent>[, <parent>...]]
```

### Examples

#### ▼ Define a Class

Lua

```
---@class Car
local Car = {}
```

#### ▼ Class Inheritance

Lua

```
---@class Vehicle
local Vehicle = {}

---@class Plane: Vehicle
local Plane = {}
```

#### ▼ Create exact class

Lua

```
---@class (exact) Point
---@field x number
---@field y number
local Point = {}
Point.x = 1 -- OK
Point.y = 2 -- OK
Point.z = 3 -- Warning
```

#### ▼ How the table Class is Implemented

Lua

```
---@class table<K, V>: { [K]: V }
```

## @deprecated

Mark a function as deprecated. This will trigger the [deprecated diagnostic](#), displaying it as ~~struck through~~.

### Syntax

```
---@deprecated
```

### Examples

#### ▼ Mark a Function as Deprecated

Lua

```
---@deprecated
function outdated() end
```

## @diagnostic

Toggle [diagnostics](#) for the next line, current line, or whole file.

### Syntax

state options:

- `disable-next-line` (Disable diagnostic on the following line)
- `disable-line` (Disable diagnostic on this line)
- `disable` (Disable diagnostic in this file)
- `enable` (Enable diagnostic in this file)

### Examples

#### ▼ Disable Diagnostic on Next Line

Lua

```
---@diagnostic disable-next-line: unused-local
```

#### ▼ Enable Spell Checking in this File

Lua

```
---@diagnostic enable:spell-check
```

## @enum

Mark a Lua table as an enum, giving it similar functionality to [@alias](#), only the table is still usable at runtime. Adding the `(key)` attribute will use the enum's keys instead of its values.

[Original Feature Request](#)

### Syntax

```
---@enum [(key)] <name>
```

### Examples

#### ▼ Define Table as Enum

Lua

```

---@enum colors
local COLORS = {
    black = 0,
    red = 2,
    green = 4,
    yellow = 8,
    blue = 16,
    white = 32
}

---@param color colors
local function setColor(color) end

setColor(COLORS.green)

```

#### ▼ Define Table Keys as Enum

Lua

```

---@enum (key) Direction
local direction = {
    LEFT = 1,
    RIGHT = 2,
}

---@param dir Direction
local function move(dir)
    assert(dir == "LEFT" or dir == "RIGHT")

    assert(direction[dir] == 1 or direction[dir] == 2)
    assert(direction[dir] == direction.LEFT or direction[dir] == direction.RIGHT)
end

move("LEFT")

```

## @field

Define a field within a table. Should be immediately following a [@class](#). As of v3.6, you can mark a field as private, protected, public, or package.

### Syntax

```

---@field [scope] <name[?]> <type> [description]

```

It is also possible to allow *any* key of a certain type to be added using the below:

```

---@field [scope] [<type>] <type> [description]

```

### Examples

#### ▼ Simple Documentation of a Class

Lua

```

---@class Person
---@field height number The height of this person in cm
---@field weight number The weight of this person in kg
---@field firstName string The first name of this person
---@field lastName? string The last name of this person
---@field age integer The age of this person

---@param person Person
local function hire(person) end

```

#### ▼ Mark Field as Private

Lua

```

---@class Animal
---@field private legs integer
---@field eyes integer

---@class Dog:Animal
local myDog = {}

---Child class Dog CANNOT use private field legs
function myDog:legCount()
    return self.legs
end

```

#### ▼ Mark Field as Protected

Lua

```

---@class Animal
---@field protected legs integer
---@field eyes integer

---@class Dog:Animal
local myDog = {}

---Child class Dog can use protected field legs
function myDog:legCount()
    return self.legs
end

```

#### ▼ Typed Field

Lua

```

---@class Numbers
---@field named string
---@field [string] integer
local Numbers = {}

```

## @generic

Generics allow code to be reused and serve as a sort of "placeholder" for a type. Surrounding the generic in backticks (`) will capture the value and use it for the type. [Generics are still WIP](#).

### Syntax

```
---@generic <name> [:parent_type] [, <name> [:parent_type]]
```

## Examples

### ▼ Generic Function

Lua

```
---@generic T : integer
---@param p1 T
---@return T, T[]
function Generic(p1) end

-- v1: string
-- v2: string[]
local v1, v2 = Generic("String")

-- v3: integer
-- v4: integer[]
local v3, v4 = Generic(10)
```

### ▼ Capture with Backticks

Lua

```
---@class Vehicle
local Vehicle = {}
function Vehicle:drive() end

---@generic T
---@param class `T` # the type is captured using `T`
---@return T # generic type is returned
local function new(class) end

-- obj: Vehicle
local obj = new("Vehicle")
```

### ▼ Array Class Using Generics

Lua

```
---@class Array<T>: { [integer]: T }

---@type Array<string>
local arr = {}

-- Warns that I am assigning a boolean to a string
arr[1] = false

arr[3] = "Correct"
```

See [Issue #734](#)

### ▼ Dictionary Class Using Generics

Lua

```
---@class Dictionary<T>: { [string]: T }

---@type Dictionary<boolean>
local dict = {}

-- no warning despite assigning a string
dict["foo"] = "bar?"

dict["correct"] = true
```

Marks a file as "meta", meaning it is used for definitions and not for its functional Lua code. It is used internally by the language server for defining the [built-in Lua libraries](#). If you are writing your own [definition files](#), you will probably want to include this annotation in them. If you specify a name, it will only be able to be required by the given name. Giving the name `_` will make it unable to be required. Files with the `@meta` tag in them behave a little different:

- Completion will not display context in a meta file
- Hovering a `require` of a meta file will show `[meta]` instead of its absolute path
- Find Reference ignores meta files

## Syntax

```
---@meta [name]
```

## Examples

### ▼ Mark Meta File

## @module

Simulates `require`-ing a file.

## Syntax

```
---@module '<module_name>'
```

## Examples

### ▼ "Require" a File

Lua

```
---@module 'http'

--The above provides the same as
require 'http'
--within the language server
```

### ▼ "Require" a File and Assign to a Variable

Lua

```
---@module 'http'
local http
```

```
--The above provides the same as
local http = require 'http'
--within the language server
```

## @nodiscard

Mark a function as having return values that **cannot** be ignored/discarded. This can help users understand how to use the function as if they do not capture the returns, a warning will be raised.

### Syntax

```
---@nodiscard
```

### Examples

#### ▼ Prevent Ignoring a Function's Returns

Lua

```
---@return string username
---@nodiscard
function getUsername() end
```

## @operator

Provides type declarations for an [operator metamethod](#).

[Original Feature Request](#)

### Syntax

```
---@operator <operation>[(param_type)]:<return_type>
```

### Examples

#### ▼ Declare \_\_add Metamethod

Lua

```
---@class Vector
---@operator add(Vector): Vector
```

```
---@type Vector
local v1
---@type Vector
local v2
```

```
--> v3: Vector
local v3 = v1 + v2
```

#### ▼ Declare Unary Minus Metamethod

Lua

```
---@class Passcode
---@operator un-:integer
```

```
---@type Passcode
local pA
```

```
local pB = -pA
--> integer
```

#### ▼ Declare \_\_call Metamethod

Lua

```
---@class URL
---@operator call:string
local URL = {}
```

## @overload

Define an additional signature for a function.

### Syntax

```
---@overload fun([param: type[, param: type...]]): [return_value[,return_value]]
```

### Examples

#### ▼ Define Function Overload

Lua

```
---@param objectID integer The id of the object to remove
---@param whenOutOfView boolean Only remove the object when it is not visible
---@return boolean success If the object was successfully removed
---@overload fun(objectID: integer): boolean
local function removeObject(objectID, whenOutOfView) end
```

#### ▼ Define Class Call Signature

Lua

```
---@overload fun(a: string): boolean
local foo = setmetatable({}, {
    __call = function(a)
        print(a)
        return true
    end,
})
```

```
local bool = foo("myString")
```

## @package

Mark a function as private to the file it is defined in. A packaged function cannot be accessed from another file.

## Syntax

```
---@package
```

## Examples

### ▼ Mark a Function as Private to a Package

Lua

```
---@class Animal
---@field private eyes integer
local Animal = {}

---@package
---This cannot be accessed in another file
function Animal:eyesCount()
    return self.eyes
end
```

## @param

Define a parameter/argument for a function. This tells the language server what types are expected and can help enforce types and provide completion. Putting a question mark (?) after the parameter name will mark it as optional, meaning nil is an accepted type. The type provided can be an [@alias](#), [@enum](#), or [@class](#), of course, as well.

## Syntax

```
---@param <name[?]> <type[|type...]> [description]
```

## Examples

### ▼ Simple Function Parameter

Lua

```
---@param username string The name to set for this user
function setUsername(username) end
```

### ▼ Parameter Union Type

Lua

```
---@param setting string The name of the setting
---@param value string|number|boolean The value of the setting
local function settings.set(setting, value) end
```

### ▼ Optional Parameter

Lua

```
---@param role string The name of the role
---@param isActive? boolean If the role is currently active
---@return Role
function Role.new(role, isActive) end
```

### ▼ Variable Number of Parameters

Lua

```
---@param index integer
---@param ... string Tags to add to this entry
local function addTags(index, ...) end
```

### ▼ Generic Function Parameter

Lua

```
---@class Box

---@generic T
---@param objectId integer The ID of the object to set the type of
---@param type `T` The type of object to set
---@return `T` object The object as a Lua object
local function setObjectType(objectID, type) end

--> boxObject: Box
local boxObject = setObjectType(1, "Box")
```

See [@generic](#) for more info.

### ▼ Custom Type Parameter

Lua

```
---@param mode string
---|`'immediate'` # comment 1
---|`'async'` # comment 2
function bar(mode) end
```

### ▼ Literal Custom Type Parameter

Lua

```
local A = 0
local B = 1

---@param active integer
---|`A` # Has a value of 0
---|`B` # Has a value of 1
function set(active) end
```

Looking to do this with a table? You probably want to use [@enum](#)

## @private

Mark a function as private to a [@class](#). Private functions can be accessed only from within their class and are **not** accessible from child classes.



## Syntax

```
---@private
```

## Examples

### ▼ Mark a function as private

```
Lua

---@class Animal
---@field private eyes integer
local Animal = {}

---@private
function Animal:eyesCount()
    return self.eyes
end

---@class Dog:Animal
local myDog = {}

---NOT PERMITTED!
myDog:eyesCount();
```

## @protected

Mark a function as protected within a [@class](#). Protected functions can be accessed only from within their class or from child classes.

## Syntax

```
---@protected
```

## Examples

### ▼ Mark a function as protected

```
Lua

---@class Animal
---@field private eyes integer
local Animal = {}

---@protected
function Animal:eyesCount()
    return self.eyes
end

---@class Dog:Animal
local myDog = {}

---Permitted because function is protected, not private.
myDog:eyesCount();
```

## @return

Define a return value for a function. This tells the language server what types are expected and can help enforce types and provide completion.

## Syntax

```
---@return <type> [<name> [comment] | [name] #<comment>]
```

## Examples

### ▼ Simple Function Return

```
Lua

---@return boolean
local function isEnabled() end
```

### ▼ Named Function Return

```
Lua

---@return boolean enabled
local function isEnabled() end
```

### ▼ Named, Described Function Return

```
Lua

---@return boolean enabled If the item is enabled
local function isEnabled() end
```

### ▼ Described Function Return

```
Lua

---@return boolean # If the item is enabled
local function isEnabled() end
```

### ▼ Optional Function Return

```
Lua

---@return boolean|nil error
local function makeRequest() end
```

### ▼ Variable Function Returns

```
Lua

---@return integer count Number of nicknames found
---@return string ...
local function getNicknames() end
```

## @see

Allows you to reference a specific symbol (e.g. function, class) from the workspace. You can also reference symbols using [markdown links instead](#).

### Syntax

```
---@see <symbol>
```

### Examples

#### ▼ Basic Usage

Lua

```
---Hovering the below function will show a link that jumps to http.get()
```

```
---@see http.get
function request(url) end
```

### @source

Provide a reference to some source code which lives in another file. When searching for the definition of an item, its @source will be used instead.

### Syntax

```
---@source <path>
```

### Examples

#### ▼ Link to file using absolute path

Lua

```
---@source C:/Users/me/Documents/program/myFile.c
local a
```

#### ▼ Link to file using URI

Lua

```
---@source file:///C:/Users/me/Documents/program/myFile.c:10
local b
```

#### ▼ Link to file using relative path

Lua

```
---@source local/file.c
local c
```

#### ▼ Link to line and character in file

Lua

```
---@source local/file.c:10:8
local d
```

### @type

Mark a variable as being of a certain type. Union types are separated with a pipe character |. The type provided can be an [@alias](#), [@enum](#), or [@class](#) as well. Please note that you cannot add a field to a class using @type, [you must instead use @class](#).

### Syntax

```
---@type <type>
```

### Examples

#### ▼ Basic Type Definition

Lua

```
---@type boolean
local x
```

#### ▼ Array Type Definition

Lua

```
---@type string[]
local names
```

#### ▼ Dictionary Type Definition

Lua

```
---@type { [string]: boolean }
local statuses
```

#### ▼ Table Type Definition

Lua

```
---@type table<userID, Player>
local players
```

#### ▼ Union Type Definition

Lua

```
---@type boolean|number|"yes"|"no"
local x
```

#### ▼ Function Type Definition

Lua

```
---@type fun(name: string, value: any): boolean
local x
```

## **@vararg**

Mark a function as having variable arguments. For variable returns, see [@return](#).

### **Syntax**

```
---@vararg <type>
```

### **Examples**

#### ▼ Basic Variable Function Arguments

Lua

```
---@vararg string
function concat(...) end
```

## **@version**

Mark the required Lua version for a function or [@class](#).

### **Syntax**

```
---@version [<|><version> [, [<|>version...]
```

Possible version values:

- 5.1
- 5.2
- 5.3
- 5.4
- JIT

### **Examples**

#### ▼ Declare Function Version

Lua

```
---@version >5.2, JIT
function hello() end
```

#### ▼ Declare Class Version

Lua

```
---@version 5.4
---@class Entry
```

Last Modified: Sep 6, 2024 6:05 PM