

**LongOS Programming Guide**

|  |  |
| --- | --- |
| Author: | Gvin |
| Version: | 1.0 |

# **1 Executive Overview**

This guide describes basics of programming for LongOS and using classes in Lua. It also contains several guidelines for those who would like to contribute his programs to the main LongOS repository (or just want to make their programming easier).

To use this document you should be familiar with such terms as “Object Oriented Programming” and “Classes”.

# **2 Table of Content**

[**1 Executive Overview** 2](#_Toc365553845)

[**2 Table of Content** 3](#_Toc365553846)

[**3 About This Guide** 4](#_Toc365553847)

[**1 OOP in Lua** 5](#_Toc365553848)

[1.1 Basics of Class Creation 5](#_Toc365553849)

[1.2 Fields and methods. Access modifiers 6](#_Toc365553850)

[1.3 Inheritance 7](#_Toc365553851)

[**2 Introduction to Application** 9](#_Toc365553852)

[2.1 Application 9](#_Toc365553853)

[2.2 Window 9](#_Toc365553854)

[**3 Application Classes Specification** 11](#_Toc365553855)

# **3 About This Guide**

In this guide you can find information about basics of programming for LongOS. It consists of several sections:

* Section 1: OOP in Lua

The basics of OOP in Lua, creation and usage of classes.

* Section 2: Introduction to Application

Some information about what does “application” means for LongOS.

* Section 3: Application Classes Specification

Full specification of application classes with detailed information.

* Section 4: Components Classes Specification

Full specification of components classes with detailed information.

* Section 5: System Classes Specification

Full specification of system classes with detailed information.

* Section 6: Guidelines

Several guidelines for those who would like to have clear and beautiful code.

* Section 7: Examples

Some examples of application creating and using some additional features.

# **1 OOP in Lua**

OOP means Object Oriented Programming. Its basics will not be described here. If you would like to know more about it search the internet.

In this section is represented the OOP programming in Lua. This includes:

* The basics of class creation in Lua
* Fields and methods. Access modifiers
* Inheritance realization

## 1.1 Basics of Class Creation

In Lua language as-it-is there are no classes at all. But there is an interesting data structure — table. With some coding it can be used to create classes and get access to almost all OOP principles.

To start creating classes in Lua you should first create function “Class” which you can find here: “LongOS/Classes/SystemClasses/ClassBase.lua”.

To create simple class you should create code like this:

local ClassExample = Class(Object, function(this)

Object.init(this, ‘ClassExample’);

end)

This class contains no fields, methods and didn’t receive any parameters in the constructor. To create an instance of in simply type:

Local instanceExample = ClassExample();

Function Class is responsible for creation of the class. First parameter of this function is parent class (for classes without parent it should be Object) and the other is special initialization function.

## 1.2 Fields and methods. Access modifiers

Fields and methods creation in classes are very similar to creation of variables and functions in standard function-style program. Access modifiers are realized through adding local prefix to variables and functions or inserting them into class table (because classes are tables).

To create private variable you should type:

local ClassWithVariables = Class(Object, function(this)

Object.init(this, ‘ClassWithVariables’);

local privateVariable = 3;

this.PublicVariable = ‘hello’;

end)

In the class above you can see the variables of two different access types: private and public. Private variables are only accessible from functions and other variables that come below their declaration in code and only inside their base class. Public variables can be accessed anywhere in code but to get access to them you must prefix their name with “**this.**”.

Methods creation is very similar:

local ClassWithMethods = Class(Object, function(this)

Object.init(this, ‘ClassWithMethods’);

local function privateMethod()

print(‘Private method.’);

end

function this:GlobalMethod()

print(‘Global method.’);

end

end)

In the class above you can see two methods: private and public. Private method, like private variable, can be used only in the code below its declaration and only inside its base class. Public methods can be used anywhere but to get access to them you must prefix their name with “**this:**”.

Protected access modifier didn’t exist in Lua.

## 1.3 Inheritance

To create class inherited from some parent class you must specify the parent’s class as the first parameter of the “Class” function.

local Animal = Class(Object, function(this, \_name)

Object.init(this, “Animal”);

local name;

function this:GetName()

return name;

end

local function constructor(\_name)

name = \_name;

end

constructor(\_name);

end)

This is the Animal class which can only store the animal’s name (receives it from the constructor) and returns it from the getter function.

local Cat = Class(Animal, function(this, \_type, \_voice)

Animal.init(this, “Cat”);

local type;

local voice;

function this:Voice()

return “Cat of type ”..type..” says “..voice;

end

local function constructor(\_type, \_voice)

type = \_type;

voice = \_voice;

end

constructor(\_type, \_voice);

end)

This is the Cat class which inherits parent class Animal. Then we can create an instance of Cat and call on it methods from Animal.

To override methods you simply must create new method with the same name as an overrided method’s name.

# **2 Introduction to Application**

Application in LongOS are much different from those in CraftOS. They are objects (as almost everything in the system) and the whole model of information processing consists of reaction on different events. The system catches events and your program just need to process them. But it is rarely needed because the standard events processing is already implemented and hidden from you.

## 2.1 Application

Applications is a complex object which contains sets of windows and threads. Also it calls event handlers on this sets according to special algorithms. To create an application you simply must create a new application object from its constructor:

local appSample = Application(“App sample”);

This is the easiest way to create an application. There are 3 constructors for Application class:

Application(string applicationName, boolean isUnique, boolean shutdownWhenNoWindows)

Application(string applicationName, boolean isUnique)

Application(string applicationName)

Any of this constructors can be used to create an application instance.

To start running the application you need to call the “Run” method. The only parameter for this method is an instance of Window class. But you can pass there “nil” value if shutdownWhenNoWindows was setted to true. About windows creation see section 2.2.

For more information see section 3.1.

## 2.2 Window

Window is the main interface part of the LongOS. It contains components, reacts on user’s actions and shows information. It is able to create event handlers for different events in a window but for most purposes there is no need of this. You can handle most of the events by using components — special elements which reacts on user’s actions or displays some information (Button, Label, Edit, etc.).

To create a window you must call its constructor:

local windowSample = Window(appSample, “Window sample”, false);

There is only one constructor fir Window class.

Window(Application parentApplication, string name, boolean isUnique)

Also there are a lot of properties that can be setted on your window. See section 3.2 for more information.

After the creation the window can be shown either as single non-modal or as modal (dialog) window. To do this you must call the “Show” (or “ShowModal”) method with the parent application as the only parameter. If you use window in “Run” function of your application you don’t need to call “Show” or “ShowModal” methods for it. They will be invoked on application starting.

# **3 Application Classes Specification**