

**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](#_Toc365630073)

[2 Table of Content 3](#_Toc365630074)

[3 About This Guide 4](#_Toc365630075)

[1 OOP in Lua 5](#_Toc365630076)

[1.1 Basics of Class Creation 5](#_Toc365630077)

[1.2 Fields and methods. Access modifiers 6](#_Toc365630078)

[1.3 Inheritance 7](#_Toc365630079)

[2 Introduction to Application 9](#_Toc365630080)

[2.1 Application 9](#_Toc365630081)

[2.2 Thread 9](#_Toc365630082)

[2.3 Window 10](#_Toc365630083)

[2.4 Components 11](#_Toc365630084)

[2.5 Menus 13](#_Toc365630085)

[3 Application Classes Specification 14](#_Toc365630086)

[3.1 Application Class 14](#_Toc365630087)

# 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](#_1_OOP_in)

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

* [Section 2: Introduction to Application](#_2_Introduction_to)

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

* [Section 3: Application Classes Specification](#_3_Application_Classes)

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](#_1.1_Basics_of)
* [Fields and methods. Access modifiers](#_1.2_Fields_and)
* [Inheritance realization](#_1.3_Inheritance)

## 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 2.3](#_2.3_Window).

For more information about application see [section 3.1](#_3.1_Application_Class).

## 2.2 Thread

You can create a parallel thread in your application and specify function for it. This can be useful for applications where some long-time actions must be done. Threads are linked to your application and will be stopped when it is closed. So in you want to create background application which will be executing some runtime actions you need to specify shutdownWhenNoWindows option to “true” on application creation.

To create a new thread you just need to create its object:

local newThread = Thread(application, workingFunction);

And then you can start the thread by calling:

newThread:Start();

For full specification see section 3.2.

## 2.3 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.).

But you can’t just create a window object from Window class because it will be only an empty window. For customizing window’s components and behavior you need to inherit it.

The sample of custom window class:

Local MyWindow = Class(Window, function(this, \_application)

Window.init(\_application, “My window”, false);

this:SetTitle(“My window”);

this:SetWidth(15);

this:SetX(3);

end)

The new window will take only 1 parameter in the constructor and will be created with title “My window”, width 15 and x position 3. In this way you can create your own event processors or add components to the window (about components see [section 2.4](#_2.4_Components)).

To add event processor you just need to override some basic function of event processing. For example:

function this:ProcessCharEvent(\_char)

…

end

This function will receive char events, directed to the window and process them in the way you like. For more event processors see window class specification.

Also there are a lot of properties that can be setted on your window. See section 3.3 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.

## 2.4 Components

Components play significant role in windows creation. They can handle most of the events and are easy to use. There are a lot of components in the LongOS. The most basic are Button, Label and Edit.

To add component to your window you need to create it first with a constructor.

local buttonSample = Button(“Sample”, nil, nil, 0, 0, “left-top”);

Then you need to add this component to the window from outside:

myWindow:AddComponent(buttonSample);

Or from the window constructor:

MyWindow = Class(Window, function(this, \_application)

Window.init(this, \_application, “My window”, false);

local buttonSample2;

local function constructor()

buttonSample2 = Button(“Sample 2”, nil, nil, 0, 0, “left-top”);

this:AddComponent(buttonSample2);

end

constructor();

end)

The button will be placed on the window and will automatically process clicks. But we also need to add some event handler on the button click. You can do this that way:

MyWindow2 = Class(Window, function(this, \_application)

Window.init(this, \_application, “My window”, false);

local buttonSample;

local function buttonSampleClick(\_sender, \_eventArgs)

-- Do something.

end

local function constructor()

buttonSample = Button(“Sample”, nil, nil, 0, 0, “left-top”);

buttonSample:AddOnClickEventHandler(buttonSampleClick);

this:AddComponent(buttonSample);

end

constructor();

end)

This code will create window class with a button “Sample”. And on clicking the button code of the “buttonSampleClick” function will be executed.

You can create other components in the same way. See section 4.1 for components specification.

## 2.5 Menus

You can also create menus for your window if you like. For now they are simple single menus with buttons. To create a menu you must create a menu object and add buttons to it. Then you need to add that menu to the window’s menus collection (using method “AddMenu(string name, Menu menu)”). To operate menu use window’s methods “OpenMenu(string name)”, “CloseMenu(string name)” and “OpenCLoseMenu(string name)”. For more information see section 4.2.

# 3 Application Classes Specification

This section contains full information about application classes. They include:

* Application Class
* Thread Class
* Window Class

## 3.1 Application Class

Application is a basic class for user’s programs. It contains windows and threads and manages event processing for them.

Constructors:

public Application(string \_applicationName, bool \_isUnique, bool \_shutdownWhenNoWindows)

public Application(string \_applicationName, bool \_isUnique)

public Application(string \_applicationName)

“\_applicationName“ — the name of application, it will be displayed in TasksManager.

“\_isUnique” — specifies that if such application is already opened it will be selected as current and not opened another instance of it. The default value for the 3-rd constructor is “false”.

“\_shutdownWhenNoWindows” — specifies that application will be closed when no windows is left. The default value for the 2-nd and the 3-rd constructors is “true”.

Methods:

public string GetName() — gets the application name.

public bool GetIsUnique() — gets if the application is unique.

public bool GetEnabled() — gets if the application is enabled (is current active application).

public void SetEnabled(bool \_value) — sets the enabled property for the application. This method is used by system classes and must not be called in programs.

public string GetId() — gets the application’s id.

public void Initialize(string \_id) — initializes the application. This method is used by system classes and must not be called in programs.

public void AddWindow(Window \_window) — adds a new window to the application. This method is called in “window:Show()” and “window:ShowModal()” so you don’t need to call them manually.

public void DeleteWindow(string \_windowId) — deletes window from the application. This method is called in “window:Close()”so you don’t need to call them manually.

public void AddThread(Thread \_thread) — adds thread to the application. This method is called in “thread:Start()” so you don’t need to call them manually.

public void RemoveThread(string \_id) — removes thread from the application. This method is called in “thread:Stop()” so you don’t need to call them manually.

public void Clear() — clear all windows and threads in the application.

public bool Contains(int x, int y) — returns if any window of the application contains specified point.

public int GetWindowsCount() — returns windows count.

public void Run(Window \_window) — starts the application with the specified window as the starting window. “\_window” can be nil if “\_shutdownWhenNoWindows” is setted to “false”.

public void Draw(VideoBuffer \_videoBuffer) — draws application to the video buffer.

public void Update() — updates the application’s state.

public void Close() — closes the application.

public void ProcessKeyEvent(int \_key) — executes **key** event processing in all windows and threads.

public void ProcessCharEvent(char \_symbol) — executes **char** event processing in all windows and threads.

public void ProcessRednetEvent(int \_id, string \_message, float \_distance, string \_side, int \_channel) — executes **rednet** event processing in all windows and threads.

public void ProcessLeftClickEvent(int \_cursorX, int \_cursorY) — executes **left click** event processing in all windows and threads.

public void ProcessRightClickEvent(int \_cursorX, int \_cursorY) — executes **right click** event processing in all windows and threads.

public void ProcessDoubleClickEvent(int \_cursorX, int \_cursorY) — executes **double click** event processing in all windows and threads.

public void ProcessLeftMouseDragEvent(int \_newCursorX, int \_newCursorY) — executes **left mouse drag** event processing in all windows and threads.

public void ProcessRightMouseDragEvent(int \_newCursorX, int \_newCursorY) — executes **right mouse drag** event processing in all windows and threads.

public void ResetDragging() — resets the dragging state of all windows. This method is used by system classes and must not be called in programs.

public void ProcessTimerEvent(int \_timerId) — executes **timer** event processing in all windows and threads.

public void ProcessRedstoneEvent() — executes **redstone** event processing in all windows and threads.

public void ProcessMouseScrollEvent(int \_direction, int \_cursorX, int \_cursorY) — executes **mouse scroll** event processing in all windows and threads.