uLua

ANT Software
Version v2.2.0
Sun Oct 29 2023

Table of Contents

uLua: Lua Modding Framework

uLua is a powerful Lua Modding framework for Unity. It enables the development of a Lua API which may be used to mod your Unity game.

uLua wraps around MoonSharp and provides an object oriented Lua Modding framework. It works by setting up an application-wide Lua context and exposing game objects to it. Objects exposed to the Lua context can then be accessed in Lua scripts, allowing users to interact with your game at runtime.

uLua includes the following features:

- Lua script execution from the Resources folder or an external directory.
- Base classes to expose your game objects and data structures to Lua.
- Event and callback systems which allow events and functions to be invoked in C# but implemented in Lua.
- The ability to organise scripts in packages which can be easily installed and removed.

This project is the result of many hours of hard work. Please support me by leaving a review on the asset store! Follow my Twitter!

What's New in v2.0

Script Packages

Script packages were introduced in 1.3.0 as a means for users to organise external scripts and control their order of execution. In this update, the script packages have been reviewed so that they can be defined in Unity's Resource folders inside a project. A package by the same name may be defined as an external script, overriding the internal definition. You may prevent a script package from being overridden using a json parameter called `AllowExternalOverride`.

These changes give more options for developers who may want to implement a large part of their game logic in Lua.

Script Execution Order

The overall script execution order was reviewed. Lua scripts are now executed in the following order:

1. Script Packages (from Resources)
2. Script Packages (from External directory)
3. Scene Script (from Resources or External directory)
4. External Scripts (general)
5. External Scripts (Scene-specific)

The design intent is for script packages to implement the base game logic and required utilities which subsequent Lua scripts may need.

Registering Events and Event Handlers

The process of registering event handlers has been streamlined. Event handlers are now registered implicitly for all objects but must follow a common naming convention. For example, an event named `GameLoaded` will implicitly call all functions named `OnGameLoaded`. In order for the implicit handler registration to work, you must register events early in your project's execution (e.g. on an object's Awake), before any of the event handlers are defined. In addition, all event handlers are automatically removed when an object is destroyed.

These changes make calling `RegisterEventHandler` and `RemoveEventHandlers` unnecessary. However, you may still use these functions to explicitly register and remove additional event handlers during runtime.

Registering events is achieved by calling the relevant API method in C#:

```
uLua.API.RegisterEvent("GameLoaded");
```

or the built-in Lua function in Lua:

```
RegisterEvent("GameLoaded");
```

uLua.API.RegisterEvent() takes a second optional parameter which determines the name of the implicit event handler function. If the second parameter is omitted, the event handler name will default to `On{EventName}` (e.g. `OnGameLoaded` in this example). An example of customising the event handler name is shown below.

```
uLua.API.RegisterEvent("GameLoaded", "GameLoadedHandler");
```

Built-in Lua functions

You may now customise the default Lua API by disabling built-in functions in the API class inspector. This allows you to select which features you want to use in your Lua code by disabling access to specific functions.

Code Review

A large part of the library has been reviewed for this version. As a result, some variable and class names have been changed, and some variables are no longer available. Efforts have been made to maximise backwards compatibility, but please check the full patch notes if you are running into issues with upgrading.

Some significant changes are listed below:

- `ResourcePath` and `UserScriptsPath` were removed from the API class. These were replaced by a single path variable named `ScriptsPath`.
- `EnableResourceScript` was renamed to `EnableObjectScript` for Lua objects.
- "User scripts" are now known as "External scripts" throughout the project.
- The low level interface `IHasLuaIndexer` has been renamed to `ILuaObject`.

Deprecated Code

All previously deprecated and obsolete code has been removed with this update. If you are upgrading from a much older version, you may run into compatibility issues with your previous scripts. This documentation provides up to date instructions on how to use uLua, however, feel free to contact me on support@antsoftware.co.uk for additional support.

Dependencies

- MoonSharp for Unity

Resources

Below are some examples of how to use uLua in Unity Engine.

- **Demo Game**: A demo game which uses uLua, and comes with its own documentation of the API and source code.

Usage Tutorial

Note: You must install the Unity MoonSharp plugin before you can use uLua. Check the *Dependencies* section above for a link to the MoonSharp plugin.

uLua consists of the following main scripts:

- **uLua.Lua**: A wrapper class providing an application-wide Lua context.
- **uLua.API**: Class that implements an event handling system and a script execution framework.
- **uLua.ExposedClass**: Class which exposes its instances to Lua. To use as a base for data structures which will be accessible in your API.
- **uLua.ExposedMonoBehaviour**: MonoBehaviour script which exposes its instances to Lua. To use as a base for game object components which will be accessible in your API.

The following tutorial is a thorough introduction to the different classes and scripts in the uLua toolkit and will walk you through performing basic tasks with uLua.

1. The API class

The uLua.API class sets up various aspects of the uLua framework such as script execution directories and the event handling system. To start using uLua, add the uLua.API script to an object in your Unity scene. For now, you can stick to the default settings and continue to the next section.

The uLua.API class implements various methods which will be useful as you develop your API. These methods are introduced in the sections below.

2. Executing Lua scripts

Lua scripts may be executed from the Resources folder of your project or from an external directory using the uLua.API class. In the following sections we will go through different approaches to executing Lua scripts.

2.1. Executing a Scene Script

A scene script is a Lua script which is executed when the Unity scene is loaded. To enable scene scripts, use the `EnableSceneScript` option of the uLua.API class on the inspector UI.

You can think of the scene script as the entry point of your game to Lua. As an example consider that your scene is named `MyGameScene` . You may create the following script to execute a print command.

MyGameScene.lua

```
print ("Hello World!");
```

To execute this script, create a file named ***MyGameScene.lua*** and place it at the designated `ScriptsPath` under the Resources folder in your project.

The `ScriptsPath` is set to the folder "Scripts" by default, which means the scene script would have to be placed in the following path:

```
Resources/Scripts/MyGameScene.lua
```

When you run your scene in the editor, you should see the message "Hello World!" output to the console. Congratulations, you have executed your first Lua script!

2.2. Executing a Scene Script Externally

The uLua.API class allows you to load the scene script from an external directory. To do so, you must use the `EnableExternalScripts` option of the uLua.API class from the inspector UI and move your scene script file to the external directory under the designated `ScriptsPath`. The scene script must be placed in a folder which is named after the scene. The directory path would be the following in this example:

```
{ExternalDirectory}/{ScriptsPath}/MyGameScene/MyGameScene.lua
```

Note: The external directory is set to Unity's `Application.persistentDataPath` by default. For more information about this directory path, check the relevant Unity documentation.

Note: The value of `ScriptsPath` is set to "Scripts" by default.

Place your script in the appropriate directory and run the scene in the editor. You should see the message "Hello World!" output to the console again.

With the `EnableSceneScript` option enabled, the uLua.API class will attempt to load the scene script from the external directory ***first***. If the scene script is not found in the external directory, it will then be loaded from the Resources folder. This allows users to override the default implementation of scene scripts by redefining them at runtime.

2.3. Manual Script Execution

Scripts can be executed manually in Lua using the built-in API functions or in C# using the uLua.API class.

In Lua, you can use the `require()` and `loadfile()` functions to execute scripts placed in the designated `ScriptsPath` under the Resources folder in your project.

For instance, to execute the following script:

```
Resources/Scripts/MyScript.lua
```

You can use one of the following commands:

```
-- require function
require("MyScript");

-- loadfile function
local MyScript = loadfile("MyScript");

-- The contents of MyScript.lua are not executed until the variable is called as a function
MyScript();
```

For more information on how to use these commands lookup the Lua programming guide.

In C#, the following two static methods are available to use in your code:

uLua.API.ExecuteFile()

uLua.API.ExecuteExternalFile()

These methods make use of the script execution framework defined by uLua.API. This means that you do not need to specify the execution directories when calling these functions. As long as a game object with your uLua.API settings is set up in your scene, you can execute external or internal Lua scripts by the path to their filename.

Check the full documentation for a list of available parameters and more information on how to use these methods.

2.4. External Scripts

In a previous section we explained how users can override a scene script. While that is useful, there are times when simply extending your scene script is all that is needed.

The uLua.API class can load additional external scripts which are executed after the scene script. To enable external scripts, use the `EnableExternalScripts` option in the inspector UI.

External scripts are indexed and loaded automatically as long as they are placed in an appropriate path under the external directory. The following paths are parsed for external scripts:

```
{ExternalDirectory}/{ScriptsPath}/
{ExternalDirectory}/{ScriptsPath}/MyGameScene/
```

Any scripts found in these path will be executed by the uLua.API class **after** the scene script is executed. Scripts placed under the `MyGameScene` folder will only be loaded when that scene is active (i.e. a scene named `MyGameScene`). All scripts placed in the designated `ScriptsPath` path will be loaded independently of which scene is loaded.

As an example, you may create the following Lua script and place it in one of those two directories:

MyUserScript.lua

```
print ("This is an external script.");
```

Run your scene and you should see the following two messages output to the console:

```
Hello World!
This is a external script.
```

2.5. Script Packages

Script packages allow users to organise scripts in groups giving better control over the order of execution of different scripts, as well as easy installation and removal of multiple scripts at once. Packages are parsed and loaded **before** the base external scripts described in the previous section, and before the scene script.

2.5.1. Script package structure and order of execution

Script packages may be loaded from the Resources folder or from the external directory. For packages to be correctly parsed, they must be placed in a folder under the designated

`ScriptPackagesPath`. For instance, a package named `MyPackage` may be located in the following external directory:

```
{ExternalDirectory}/{ScriptPackagesPath}/MyPackage/
```

or in the following directory under the Resources folder:

```
Resources/{ScriptPackagesPath}/MyPackage/
```

Note: The value of `ScriptPackagesPath` is set to "Scripts/Packages" by default.

Each package folder ***must*** contain a json file with the same name which acts as the table of contents for the package. The following is an example of the json file for the `MyPackage` package:

MyPackage.json

```
{
    "Title": "Script Package Title",
    "Description": "This is a description of the script package.",
    "Version" : "v0.0.1"
}
```

Order of execution. Script packages are executed in alphabetical order based on the folder name. To organise script packages in levels of execution, you can add a number followed by a dash at the beginning of the folder name. For example, in the folder structure listed below, `MyPackage` will always be executed before OtherPackage.

```
{ExternalDirectory}/{ScriptPackagesPath}/0-MyPackage/
{ExternalDirectory}/{ScriptPackagesPath}/1-OtherPackage/
```

Note: All characters prior to the first dash in the folder name when parsing for the name of the script package.

If packages are loaded from the Resources folder you have to instead use the filename of the json file to achieve the same effect. For instance:

```
Resources/{ScriptPackagesPath}/MyPackage/0-MyPackage.json
Resources/{ScriptPackagesPath}/OtherPackage/1-OtherPackage.json
```

2.5.2. Package contents and order of execution

The properties listed in the previous example of a json file are the title, description, and version of the package which can be used for display purposes. Here, we cover how to define the contents and dependencies of a package.

The package json file allows control of the execution order of scripts within packages. As an example, see the contents tag in the file below:

MyPackage.json

```
{
    "Title": "Script Package Title",
    "Description": "This is a description of the script package.",
    "Version" : "v0.0.1",
    "LoadAllFiles": true,
    "LoadOnDemand": false,
    "Contents": [
        "File1",
        "File0"
    ]
}
```

In this example we introduce three new properties: `Contents` , `LoadAllFiles` , and `LoadOnDemand` .

Contents: The files listed in the contents tag will always be executed in the order they appear in the list. As a result, the contents tag is used to control the script order of execution within the package. Note that the file extension must be omitted from names in the contents list.

LoadAllFiles: The `LoadAllFiles` property determines whether all scripts found in the package folder will be executed (`"LoadAllFiles": true`) or only those specified in the contents list (`"LoadAllFiles": false`). `LoadAllFiles` defaults to true if omitted.

LoadOnDemand: The `LoadOnDemand` property determines the execution behaviour of the package. If set to true, the package will not be executed until explicitly requested. Otherwise, it will be loaded when the scene is loaded. `LoadOnDemand` defaults to false if omitted.

2.5.3. Dependencies

When a script package depends on another package to function correctly, then the order of execution of the two packages becomes very important. You may define a list of package names as dependencies which, if present, will be loaded before the package contents. An example json file is shown below:

MyPackage.json

```
{
    "Title": "Script Package Title",
    "Description": "This is a description of the script package.",
    "Version" : "v0.0.1",
    "Dependencies": [
        "OtherPackage"
    ]
}
```

In this case, the API will look for and execute the package named `OtherPackage` when loading `MyPackage` .

If a dependency is not found, the script package will not be executed.

2.5.4. Package API Functions

The Lua API maintains a list of installed packages which is accessible for reference and for loading packages on demand. This data is accessible through the C# API class:

uLua.API.GetPackage();

uLua.API.GetPackageCount();

uLua.API.GetPackageName();

uLua.API.LoadPackage();

and as globals in the Lua context:

```
GetPackage();
GetPackageCount();
GetPackageName();
LoadPackage();
```

Check the full documentation for a list of available parameters and more information on how to use these methods.

3. Using the Event System

In the previous section we went over the Lua script execution framework of uLua.API. Another main feature of uLua.API is its event system. Events are very useful when implementing game behaviour, and uLua allows you to invoke game events and handle them by implementing event handlers in Lua.

3.1. Invoking Events

Events are invoked by the uLua.API class. The related method is static, which means it can be executed from anywhere in your project. For instance, to invoke an event named `PlayerHealthChanged` you can use the following command anywhere in your scene:

```
uLua.API.Invoke("PlayerHealthChanged");
```

When invoking an event, you may pass any arguments as optional parameters following the event name. For instance:

```
uLua.API.Invoke("MyEventWithParameters", 5, 1, "text");
```

You may also invoke an event in a Lua script with the same syntax:

```
Invoke("MyEventWithParameters", 5, 1, "text");
```

Any arguments passed to the uLua.API.Invoke() method are automatically handled in Lua and can be included in the event handler definition.

For instance:

```
function MyHandlerWithParameters(arg1, arg2, arg3)
    -- Do stuff with arg1, arg2, arg3
end
```

You can name your game events whatever you like, however, the same event name must be used in the event handler registration.

3.2. Registering Events

All events must be registered before any event handlers are defined, and before events are invoked. You must register events early in your project's execution, but ***after*** uLua.API's Awake function is executed.

Registering events is achieved by calling the relevant API method in C#:

```
uLua.API.RegisterEvent("PlayerHealthChanged");
```

or the built-in Lua function in Lua:

```
RegisterEvent("PlayerHealthChanged");
```

Event handlers will be registered implicitly for all objects but must follow a common naming convention. For example, an event named `PlayerHealthChanged` will implicitly call

all functions named `OnPlayerHealthChanged`. In addition, all such event handlers are automatically removed when an object is destroyed.

uLua.API.RegisterEvent() takes a second optional parameter which determines the name of the implicit event handler function. If the second parameter is omitted, the event handler name will default to `On{EventName}` (e.g. `OnPlayerHealthChanged` in this example). An example of customising the event handler name is shown below.

```
uLua.API.RegisterEvent("PlayerHealthChanged", "PlayerHealthChangedHandler");
```

3.3. Registering Event Handlers

Additional custom event handlers can be registered and removed manually using the uLua.API class. The related methods are static, which means they can be executed from anywhere in your project. To register an event handler for the `PlayerHealthChanged` event you can use the following command in C#:

```
uLua.API.RegisterEventHandler("PlayerHealthChanged", "HandlePlayerHealthChanged");
```

This command registers the `HandlePlayerHealthChanged` global Lua function to be called when the `PlayerHealthChanged` event is invoked. The event handler `HandlePlayerHealthChanged` must be defined ***before*** the uLua.API.RegisterEventHandler() command is called, otherwise invoking the `PlayerHealthChanged` event will have no effect.

Note: Keep in mind that event handlers can be registered as members of other objects in Lua by using an optional third argument. For the sake of simplicity, we use a global event handler in this step of the tutorial.

You can now use your scene script to implement the `HandlePlayerHealthChanged` function.

MyGameScene.lua

```
function HandlePlayerHealthChanged()
    -- Do stuff
end
```

The methods to register and remove event handlers are also available in Lua. As a result the same event handler may instead be registered in your Lua scene script using a similar syntax. Returning to our previous example:

MyGameScene.lua

```
function HandlePlayerHealthChanged()
    -- Do stuff
end

RegisterEventHandler("PlayerHealthChanged", "HandlePlayerHealthChanged");
```

3.4. Removing Event Handlers

Any additional custom event handlers may be removed by calling the following command in C#:

```
uLua.API.RemoveEventHandlers();
```

Or the following equivalent command in a Lua script:

```
RemoveEventHandlers();
```

These commands remove all *global* event handlers.

Note: Again, keep in mind that event handlers are not exclusively globals. Event handlers for a specific object are removed by using an optional argument. For the sake of simplicity, we use global event handlers in this step of the tutorial.

3.5. The SceneLoaded / SceneUnloaded Events

The uLua.API class invokes a few events by default, including the `SceneLoaded` and `SceneUnloaded` events.

The `SceneLoaded` event is invoked after the scene script is executed, and before any external scripts are executed. The `SceneUnloaded` event is invoked when a scene is unloaded.

You may define an event handler for these events as you would for any other event. For instance, you could add the following event handlers to set up your scene:

MyGameScene.lua

```
function OnSceneLoaded()
    -- Do other stuff
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

function HandlePlayerHealthChanged()
    -- Do stuff
end

RegisterEventHandler("PlayerHealthChanged", "HandlePlayerHealthChanged");
```

4. Exposing Objects to Lua

In the previous two sections we have shown how scripts can be structured to handle events invoked in your Unity scene. However, the API that we have set up so far has no game objects exposed to it that would allow us to implement game behaviour.

To expose objects to the Lua API, uLua provides two approaches: the uLua.ExposedClass and uLua.ExposedMonoBehaviour classes. Any script that inherits these classes will automatically expose its instances as objects of the API.

When exposing game objects to the API, the uLua framework uses an object-oriented approach. Each game object is exposed to the API with a unique Lua handle. As we will explore in the following sections, this becomes important when developing your API and structuring your API object scripts.

4.1. The ExposedMonoBehaviour script

uLua.ExposedMonoBehaviour is a MonoBehaviour script. It may be used as a component similarly to MonoBehaviour, but has the added functionality of being available in Lua.

When a class inherits uLua.ExposedMonoBehaviour, all its public members are accessible in Lua. This makes uLua.ExposedMonoBehaviour a good base for developing your API objects.

As an example, let's assume you want to expose your `Player` object to Lua. For simplicity, let's say the Player class has only one member: its health. You may start by defining a `Player` script which inherits ExposedMonoBehaviour:

Player.cs

```
using uLua;
```

```
public class Player : ExposedMonoBehaviour {
    public int Health = 100;
}
```

You may now attach the `Player` script to a game object in your scene. As an example let's name that object `MyPlayer`.

By default, the uLua.ExposedMonoBehaviour class will use the game object's name to generate a handle in Lua. This means that a game object named `MyPlayer` in your Unity Scene will be accessible by the same name in Lua. You can specify a different name for your ExposedMonoBehaviour objects by using the `Name` field in the Unity inspector or in a script. This allows you to separate the game object name from the ExposedMonoBehaviour name. In addition, it allows multiple ExposedMonoBehaviour components to be attached to the same game object, while being accessed as separate objects in Lua. For this tutorial, you should leave the `Name` field blank.

Note: Game objects may not include spaces or special characters in their name if they are to be exposed to Lua. Object names must follow Lua naming conventions for variables.

By default, the game object containing the `Player` script will be exposed to the API when the object's `Start()` method is called. This behaviour may be changed by setting the `ExposeOn` parameter of uLua.ExposedMonoBehaviour to `Awake`, `SceneLoaded`, `Manual` or to `None`, allowing you to expose objects manually by using uLua.API.Expose<T>(). Finally, you may override the `OnExpose()` method in C# to run custom code when an object is exposed to Lua. For example:

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour {
    public int Health = 100;

    protected override void OnExpose() {
        print ("Player object exposed to Lua!");
    }
}
```

The `Player` script we defined above has a public field named Health. Building on the *MyGameScene.lua* script from the previous section, the following script shows how the object `MyPlayer` and its member `Health` are now accessible in Lua:

MyGameScene.lua

```
function HandlePlayerHealthChanged()
    -- Do stuff
end

function OnSceneLoaded()
    print (MyPlayer.Health);
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "HandlePlayerHealthChanged");
```

The above script will output the health of the object `MyPlayer` to the console when the scene is loaded.

It is important to note that the Lua object `MyPlayer` is a reference, not a copy of the Unity object. This means that if the `Health` member is altered in Lua, its value in Unity will also change.

To make use of the `PlayerHealthChanged` event, we can build on the `Player` class design. In the modification below, we implement the `Health` variable as a property with a public get, and add a public method named `Damage()`. The `Damage()` method changes the value of `_Health` and invokes the `PlayerHealthChanged` event.

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour {
    public void Damage(int Damage) {
        Health -= Damage;

        API.Invoke("PlayerHealthChanged");
    }

    public int Health { get; }
}
```

You may then use the `Damage()` method in Lua, and also make use of the invoked event `PlayerHealthChanged` as shown below:

MyGameScene.lua

```
function HandlePlayerHealthChanged()
    print (MyPlayer.Health);
end

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(5);
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "HandlePlayerHealthChanged");
```

The above example covers the most basic usage of this toolkit. It is recommended you experiment until you find a design that works for your project. For instance, if your Player object is comprised of several components, and you need properties of these components to be accessible in Lua, then you may have to write part of your `Player` script as a wrapper class, implementing getter and setter properties for its components as necessary for your Lua API. Alternatively, you may implement multiple components as an ExposedMonoBehaviour and expose them under a different name in Lua.

4.2. The ExposedClass script

uLua.ExposedClass is intended for data structures which need to be exposed to the Lua API. It is similar to uLua.ExposedMonoBehaviour, however, the key difference is that uLua.ExposedClass is a simple C# class instead of a MonoBehaviour script.

As an example, you may use uLua.ExposedClass to describe a weapon item in your game. A simple class definition is the following:

Weapon.cs

```
using uLua;

public class Weapon : ExposedClass {
    public int Damage = 0;

    // Public constructor
    public Weapon(string Name, LuaMonoBehaviour Context = null, bool ExposeOnInit = true,
bool EnableObjectScript = false): base(Name, Context, ExposeOnInit, EnableObjectScript)
{
    }
}
```

Classes which inherit uLua.ExposedClass need to implement the public constructor as shown in the example code above. This is because the base constructor is used to expose the object to the API and to initialise its `Name` and `Context` properties.

Note: We have not yet covered what the `Context` object is, or how to use the `EnableObjectScript` parameter. These will be explained at a later section.

Therefore, to expose an instance of `Weapon` to Lua, all you need to do is instantiate it with the `new` keyword. In the following example, we instantiate a `Weapon` named `Sword` in the `Start()` method of the `Player` script.

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour {
    public void Damage(int Damage) {
        Health -= Damage;

        API.Invoke("PlayerHealthChanged");
    }

    public int Health { get; }

    private void Start() {
        Weapon Sword = new Weapon("Sword");
    }
}
```

As long as this object instantiation is made somewhere in your code, you may access the `Sword` instance of the `Weapon` class as a global in Lua. Again, building on the *MyGameScene.lua* script from the previous section:

MyGameScene.lua

```
Sword.Damage = 5;

function HandlePlayerHealthChanged()
    print (MyPlayer.Health);
end

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(Sword.Damage);
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "HandlePlayerHealthChanged");
```

In this example we set the value of the `Sword.Damage` property to 5, and use that as a parameter when calling the method `Damage()` .

Similarly to uLua.ExposedMonoBehaviour, the class definition given above will expose all `Weapon` objects to Lua when they are instantiated. This behaviour may be disabled by setting the optional `ExposeOnInit` parameter of the uLua.ExposedClass constructor to `false` , allowing you to expose objects manually by using uLua.API.Expose<T>(). An example of a modified constructor is shown below.

Weapon.cs

```
using uLua;

public class Weapon : ExposedClass {
    public int Damage = 0;

    // Public constructor
    public Weapon(string Name, LuaMonoBehaviour Context = null): base(Name, Context,
false) {
    }
}
```

4.3. Using the Object Context

In the previous two sections we went over the basics of using uLua.ExposedClass and uLua.ExposedMonoBehaviour classes to expose objects to your API. The `Context` object is an important feature of these classes.

The `Context` object allows you to set a hierarchy for objects exposed to your API and implements a parent-child relationship. If a `Context` object is not specified, a Lua object is defined as a global by default. This is what we have done so far.

If a `Context` object is specified, a Lua object is defined as a field of the `Context` object. This feature may be used for organisation purposes as your API grows in scope.

As an example, we return to the `Player` and `Weapon` scripts from the previous section:

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour {
    public void Damage(int Damage) {
        Health -= Damage;

        API.Invoke("PlayerHealthChanged");
    }

    public int Health { get; }

    private void Start() {
        Weapon Sword = new Weapon("Sword", this);
    }
}
```

In this case we have changed the instantiation of `Weapon` to include a second parameter which assigns the `Player` object as the context of `Sword`. This is indicated by the second parameter (keyword `this`) in the `Weapon()` constructor.

To access the non-global `Sword` object in Lua, you may use the syntax `MyPlayer.Sword`, as shown below:

MyGameScene.lua

```
MyPlayer.Sword.Damage = 5;

function HandlePlayerHealthChanged()
    print (MyPlayer.Health);
end

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(MyPlayer.Sword.Damage);
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "HandlePlayerHealthChanged");
```

Here we have added a command to set the `Sword.Damage` property to 5 for the `Sword` object which is in the context of our `MyPlayer` object.

The `Context` must be of type uLua.LuaMonoBehaviour, which is the base of uLua.ExposedMonoBehaviour. As a result, the `Context` must be a game object, and cannot be a uLua.ExposedClass object. To set the context of a uLua.ExposedMonoBehaviour game object, you can use its public member `Context` or the inspector UI.

4.4. Using Object Callback Functions

In this section we will go over invoking and implementing callback functions for your API objects. This is a feature of both uLua.ExposedClass and uLua.LuaMonoBehaviour which allows callback methods to be invoked in Unity and handled in Lua.

The callback system is similar to the event handling system. The key difference is that callbacks are object-specific. In contrast, event handlers may be defined as globals or as members of any object.

Invoking a callback function is as simple as invoking an API event. It is achieved using the methods uLua.ExposedClass.InvokeLua() and uLua.ExposedMonoBehaviour.InvokeLua().

All instances of uLua.ExposedClass and uLua.ExposedMonoBehaviour invoke the "OnLoad" and "OnExit" callbacks by default. "OnLoad" is invoked after the object is first exposed to the API. "OnExit" is invoked when an object is destroyed.

You may invoke these callbacks as needed in your code, and you may invoke custom callbacks as well. In the following piece of code I have invoked the `OnDamageTaken` callback in the `Player` script.

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour {
    public void Damage(int Damage) {
        Health -= Damage;

        API.Invoke("PlayerHealthChanged");
        InvokeLua("OnDamageTaken");
    }

    public int Health { get; }

    public void Start() {
        Weapon Sword = new Weapon("Sword", this);
    }
}
```

You may implement this callback function anywhere in Lua. The callback function must be a member of a `Player` object, because it is invoked by the `Player` script. In this case, our `Player` object is named `MyPlayer`, so we implement the `OnDamageTaken` callback function as member of `MyPlayer`.

MyGameScene.lua

```
MyPlayer.Sword.Damage = 5;

function HandlePlayerHealthChanged()
    print (MyPlayer.Health);
end

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(MyPlayer.Sword.Damage);
end

function MyPlayer:OnDamageTaken()
    print (self.Health);
```

```
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "HandlePlayerHealthChanged");
```

In Lua, the ":" syntax in `MyPlayer:OnDamageTaken()` is used as syntactic sugar to mimic object oriented design. Defining the callback function as `MyPlayer:OnDamageTaken()` allows use of the `self` keyword to access members of this object.

In our example, the event `PlayerHealthChanged` and the callback function `OnDamageTaken` achieve the same effect. There are operational differences between the two so you must choose whether to use a callback function or an event on a case by case basis. The main differences between the two options are the following:

- Event handlers may be defined as globals or as a member of any object. Callback functions are object-specific.
- To implement a callback function, you must know the Lua handle of an object (`Context.Name`) or have a reference to the object, e.g. returned from a function.

4.5. Using Object Resource Scripts

Another feature of the uLua.ExposedClass and uLua.ExposedMonoBehaviour classes is the ability to execute a Lua script for each object that is exposed to the Lua API. You may use this feature to implement base functionality for your game objects which cannot be modified by external scripts.

- To use this feature for instances of uLua.ExposedMonoBehaviour, you must enable the `EnableObjectScript` option in the inspector UI of a certain object.
- To use this feature for instances of uLua.ExposedClass, you must use the `EnableObjectScript` parameter in the constructor definition of that class. For an example refer to previous implementation of a Weapon class in section 4.2.

When the object resource script is enabled, uLua will execute a Lua script by the name of the object after it is exposed to the API. Object scripts are executed from the `ScriptsPath` under the resource directory in your project.

For instance, for a `Player` object named `MyPlayer`, you may place a resource script in the following path:

```
Resources/Scripts/MyPlayer.lua
```

Note: This is assuming that the `ScriptsPath` is set to its default value of "Scripts".

4.6. Overriding, and Hiding Objects and Members in Lua

4.6.1. Overriding Methods

Allowing users to override a method in a modding framework enables mods to significantly change object behaviour. By default, methods defined in `ExposedClass` and `ExposedMonoBehaviour` scripts may not be overridden in Lua.

To allow a method to be overridden in Lua, you may use the the `[AllowLuaOverride]` attribute. The use of this feature is limited to method calls made in a Lua script. If an overridden method is called in C#, its original implementation will be called instead.

An example is shown below for the `Damage()` method.

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour {
    [AllowLuaOverride]
    public void Damage(int Damage) {
        Health -= Damage;

        API.Invoke("PlayerHealthChanged");
        InvokeLua("OnDamageTaken");
    }

    public int Health { get; set; }

    public void Start() {
        Weapon Sword = new Weapon("Sword", this);
    }
}
```

You may now use a Lua script (e.g. the object resource script) to override the implementation of that method.

MyPlayer.lua

```
function MyPlayer:Damage(Value)
    self.Health = self.Health - Value;

    Invoke("PlayerHealthChanged");
end
```

4.6.2. Hiding Objects from External Scripts

It is often useful to hide certain objects of the API from users. To achieve this, you may use the `ExternalBlacklist` feature. This is available in the API class through the Unity inspector. Adding the name of a global object to the `ExternalBlacklist` list will make it unavailable to external scripts at runtime.

This feature may be used to hide objects of the API from users, while keeping them available to use in-engine.

4.6.3. Hiding Members

Sometimes it is unsafe to expose certain class members to your Lua API. To prevent a public member from being exposed to Lua, you may add the relevant MoonSharp attribute to a class:

```
[MoonSharpHideMember("MemberName")]
```

For instance, if you wanted to hide the `Awake()` method of the `Player` script described above, you may use the following syntax:

Player.cs

```
using MoonSharp.Interpreter;
using uLua;

[MoonSharpHideMember("Start")]
public class Player : ExposedMonoBehaviour {
    public void Damage(int Damage) {
        Health -= Damage;
        API.Invoke("PlayerHealthChanged");
        InvokeLua("OnDamageTaken");
    }

    public int Health { get; set; }

    public void Start() {
        Weapon Sword = new Weapon("Sword", this);
    }
}
```

This would prevent the `Start()` method from being called within the Lua API. This could also be achieved by defining the `Awake()` method as `protected` or `private`. However, that would not be an option in cases where a specific member must be public so that other scripts in your project can have access to it.

5. Saving and Loading Object Data

The uLua.API script allows you to save variables of a Lua API object into a file which can then be loaded again at runtime. This feature may be used to save settings or other information and load it seamlessly back into your API in-between playthroughs.

To save data and load for a Lua object, you can use the following commands anywhere in your scene:

```
uLua.API.SaveData();
uLua.API.LoadSavedData();
```

These methods take 2 arguments:

- *LuaMonoBehaviour Object* : The object for which data will be saved or loaded.
- *string Index* : (Optional) The index of the data to be saved or loaded. Default value is "SaveData".

For more information, check out the full documentation below:

uLua.API.SaveData();

uLua.API.LoadSavedData();

You may save any number of variables, however, the save data must be in the format of a Lua table. The data with the index "SaveData" is saved by default, unless specified otherwise by the second argument of `SaveData()` and `LoadSavedData()`.

For instance, the following Lua table could be used to store the position of the `MyPlayer` object we defined in section 4.

```
MyPlayer.SaveData = {
    ["X"] = 10,
    ["Y"] = -10,
}
```

The members of the table determine what information will be saved. Nested tables are not supported.

The SaveData table may be defined anywhere in Lua:

- in a scene script,
- in an object script, or
- in external scripts.

You can then call the the saving/loading methods as required in your code. For instance, you may use Unity's `Start()` and `OnDestroy()` methods. Returning to the previous example of a `Player` script:

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour {
    public void Damage(int Damage) {
        Health -= Damage;

        API.Invoke("PlayerHealthChanged");
        InvokeLua("OnDamageTaken");
    }

    public int Health { get; set; }

    public void Start() {
        Weapon Sword = new Weapon("Sword", this);
    }

    protected override Start()
        base.Start();

        API.LoadSavedData(this);
    end

    protected override void OnDestroy() {
        base.OnDestroy();

        API.SaveData(this);
    }
}
```

These methods are also available in the Lua API as globals with the same arguments:

```
SaveData();
LoadSavedData();
```

and may be implemented similarly in `OnLoad` and `OnExit` callbacks. For example:

MyUserScript.lua

```
function MyPlayer:OnLoad()
    LoadSavedData(self);
end

function MyPlayer:OnExit()
    SaveData(self);
end
```

Keep in mind that in both cases the `MyPlayer.SaveData` table always has to be updated with the current values of X and Y before calling the `SaveData()` method. The Lua callback `OnExit` is a suitable place to do this in both cases.

6. Message, warning, and error logging

You may use the following commands for message logging and error reporting in Lua:

```
print("Hello World!");
LogWarning("Hello World!");
LogError("Hello World!");
```

These commands are linked to Unity's corresponding `Debug.Log()`, `Debug.LogWarning()`, and `Debug.LogError()`, which means that any messages printed in Lua will also be printed to the console in the Unity editor.

Finally, these commands trigger the following events in Lua:

```
LuaMessageLogged
LuaWarningLogged
LuaErrorLogged
```

The first parameter passed in an event handler for these events contains the string that was logged. For instance:

```
function OnLuaMessageLogged(Text)
    -- 'Text' contains the string "Hello World!"
    -- Do stuff
end

print("Hello World!");
```

7. Visual Studio Code Debugger

uLua supports the Visual Studio Code debugger for MoonSharp. To use the debugger you must install Visual Studio Code and download the MoonSharp Debug extension from the Visual Studio marketplace. The MoonSharp Debug extension is not actively being developed, however, it works if set up correctly.

To set up the debugger follow the instructions below:

- Make sure that your MoonSharp installation includes 'MoonSharp.Debugger.VsCode.dll'. The MoonSharp package on the Unity Asset Store already includes this.
- Install Visual Studio Code and the MoonSharp Debug extension from the Visual Studio marketplace.
- Create a workspace in Visual Studio Code with your Lua scripts. Your workspace can include both resource scripts and external scripts from different folders.
- Create a file called 'launch.json' and place it inside the '.vscode' folder in your workspace directory. Copy the contents listed below to your 'launch.json' file.

.vscode/launch.json

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "debugServer" : 41912,
            "name": "MoonSharp Attach",
            "type": "moonsharp-debug",
            "request": "attach"
        }
    ]
}
```

Visual Studio Code Debug uses the 'launch.json' file to attach the debugger to your Lua scripts, so it is critical that it is set up correctly. **It is worth noting that official documentation on the structure of the 'launch.json' file is incorrect, so make sure you use the structure given here.**

- Enable the Visual Studio Code Debugger in your Unity scene by selecting the option 'Enable Debugger' on the API object inspector.
- Start your scene in the Unity editor or execute your built game.

- Attach the debugger in Visual Studio Code by selecting 'Run -> Start Debugging' on the Visual Studio Code menu or pressing (F5). If the debugger is attached correctly, you will see Lua scripts listed in the 'Debug Console' window in Visual Studio Code as they are executed.

At this point you will be able to use breakpoints, pause/continue execution, and step through your Lua scripts in Visual Studio Code. The debugger is not perfect, but it works.

The debugger relies on finding the Lua script files in a directory in order to apply breakpoints. If a file is not found during execution (e.g. a resource file in a compiled version may not be present on the hard drive), MoonSharp creates a temporary copy of the script which you can step through. The location of these temporary files is listed in the debug console as they are executed. You may still use breakpoints in the temporary scripts, but you must use the file generated by MoonSharp.

For Lua scripts located in Unity resource folders, it is important that the `MainResourcesFolder` variable in your API object is set correctly in the inspector. uLua uses this path to locate the resource scripts for debugging, so it is advised that you place all your scripts in the same resources folder. If your Lua scripts are split between multiple resource folders within your project, the debugger will not be able to locate all of them and will create temporary copies of the scripts for debugging.

8. Further Support

This covers a large portion of the features in uLua! I hope that this tutorial together with the documentation are enough to get you started on your API development. However, for any further questions do not hesitate to contact support@antsoftware.co.uk.

I have put together a demo paddle game project which utilises uLua, and which comes with its own documentation of the API and source code. You may find it here.

This project is the result of many hours of hard work. Please support me by leaving a review on the asset store! Also, follow my Twitter!

Namespace Index

Namespace List

Here is a list of all documented namespaces with brief descriptions:

Hierarchical Index

Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Class Index

Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Namespace Documentation

uLua Namespace Reference

Namespace containing the uLua project.

Classes

- class AllowLuaOverrideAttribute
  Custom attribute which allows a userdata method to be overriden in a Lua script. Intended for use with ExposedClass and ExposedMonoBehaviour.
- class API
  This script sets up the API framework to your Unity scene.
- class Events
  Static container class for registered events.
- class ExposedClass
  Class structure exposed to Lua. You should use this class as a base for your API data structures.
- class ExposedMonoBehaviour
  MonoBehaviour script exposed to Lua. You should use this class as a base for your API game objects.
- interface ILuaObject
  Interface used to implement a fully indexed Lua object.
- class IndexedUserDataDescriptor
  Custom user data descriptor to more accurately implement Lua syntax in the Moonsharp interpreter. Makes use of the ILuaObject interface.
- class Lua
  Wrapper class which streamlines use of the MoonSharp Lua context.
- class LuaClass
  Class structure which establishes a Lua object framework. For internal use.
- class LuaMonoBehaviour
  MonoBehaviour script which establishes a Lua object framework. For internal use.
- class ScriptLoader
  A script loader to load scripts from assets in Unity from within the Lua context. Scripts will be loaded from a subdirectory of Assets/Resources.
- class ScriptPackage
  A class to describe Lua packages.
- class ScriptPackageJson
  A simple structure used to load ScriptPackage information from a json file using Unity's JsonUtility.

Enumerations

- enum ExposeOn
  Indicates when to expose an object to Lua.

Detailed Description

Namespace containing the uLua project.

Class Documentation

uLua.AllowLuaOverrideAttribute Class Reference

Custom attribute which allows a userdata method to be overriden in a Lua script. Intended for use with ExposedClass and ExposedMonoBehaviour.

Detailed Description

Custom attribute which allows a userdata method to be overriden in a Lua script. Intended for use with ExposedClass and ExposedMonoBehaviour.

You may add the [AllowLuaOverride] attribute to any method of a class which uses the IndexedUserDataDescriptor structure.

uLua.API Class Reference

This script sets up the API framework to your Unity scene.

Public Member Functions

- ScriptPackage GetPackage (string PackageName)
  Returns the ScriptPackage object for the specified package name. If the package is not found, returns a null value.

- int **GetPackageCount** ()
  Returns the number of packages found under the script package path.

- string **GetPackageName** (int i)
  Returns the name of the package at the specified index.

- bool LoadPackage (string PackageName)
  Public function used to load a ScriptPackage on demand.

Static Public Member Functions

- static bool ExecuteExternalFile (string File, bool overwriteExisting=false)
  Executes a script from the external directory.

- static bool ExecuteFile (string File, bool overwriteExisting=false)
  Executes a script from the resource directory.

- static void Expose< T > (T Object, LuaMonoBehaviour Context=null)
  Exposes an object to Lua.

- static void Invoke (string Event, params object[] args)
  Invokes a game event.

- static void Log (string String)
  Logs a message to the console.

- static void LogError (string String)
  Logs an error to the console.

- static void LogWarning (string String)
  Logs a warning to the console.

- static void LoadSavedData (LuaMonoBehaviour Object, string Index="SaveData")
  Loads table data saved for the specified object.

- static void RegisterEvent (string Event, string HandlerName="")
  Registers an event to the API.

- static bool RegisterEventHandler (string Event, string HandlerName, ILuaObject Context=null)
  Registers an event handler to be called when an event is invoked.

- static void RemoveEventHandlers (LuaMonoBehaviour Context)
  Removes all event handlers registered in a specific context.

- static void RegisterIndexedType (Type Type)
  Registers a type that implements the ILuaObject interface to Lua.

- static void RegisterType< T > ()
  Registers a type to Lua.

- static void SaveData (LuaMonoBehaviour Object, string Index="SaveData")
  Saves table data from the specified object.

- static bool TryRegisterEventHandler (string HandlerName, ILuaObject Context=null)
  Attempts to register an event handler for a registered event.

Static Public Attributes

- static string **ExternalDirectory** = ""
  Directory for external Lua scripts. The path is determined at run-time.

- static string **SaveDataPath** = ""
  Path for user save data under the external directory.

- static string **ScriptsPath** = ""
  Path for user scripts under the external directory.

- static string **MainResourcesFolder** = ""
  Path to the main resources folder for scripts. Used by the VS Code debugger to find the Lua scripts.

- static string **ScriptPackagesPath** = ""
  Path for script packages under the external directory.

- static string **LuaScriptsExtension** = ""
  File extension for all external Lua scripts.

Detailed Description

This script sets up the API framework to your Unity scene.

Includes methods to execute Lua scripts, expose game objects to the API, and invoke events. Inherits from MonoBehaviour. Intended use is to add to an object in the scene and set up for use.

This script invokes the `SceneLoaded` event when the scene is loaded.

Member Function Documentation

static bool uLua.API.ExecuteExternalFile (string *File*, bool *overwriteExisting* = `false`)`[inline]`, `[static]`

Executes a script from the external directory.

Parameters

File	The Lua file to execute. The file extension must be omitted.
overwriteExisting	(Optional) If true, the method will overwrite any previous script with the same filename. Otherwise, any previous script will be called from a hash table.

static bool uLua.API.ExecuteFile (string *File*, bool *overwriteExisting* = `false`)`[inline]`, `[static]`

Executes a script from the resource directory.

Parameters

File	The Lua file to execute. The file extension must be omitted.
overwriteExisting	(Optional) If true, the method will overwrite any previous script with the same filename. Otherwise, any previous script will be called from a hash table.

static void uLua.API.Expose< T > (T *Object*, LuaMonoBehaviour *Context* = `null`)`[inline]`, `[static]`

Exposes an object to Lua.

Called internally by uLua.ExposedMonoBehaviour.Start(). Can be called manually at an earlier point or for objects which have `ExposeOnStart` disabled.

Parameters

Object	The object to be exposed to Lua. The generic type `T` must implement the ILuaObject interface.
Context	The Lua context of the object. If null, the object will be exposed as a global.

Type Constraints

T : *class*
T : *ILuaObject*

ScriptPackage uLua.API.GetPackage (string *PackageName*)`[inline]`

Returns the ScriptPackage object for the specified package name. If the package is not found, returns a null value.

Parameters

PackageName	The name of the package.

static void uLua.API.Invoke (string *Event*, params object[] *args*) [inline], [static]

Invokes a game event.

This method is exposed to Lua as a global and may be called from a Lua script or within C#. Events should registered AFTER Unity's Awake call to prevent unexpected behaviour (e.g. on Unity's scene loaded or start events). Events may also be registered when an object is exposed to Lua by overriding uLua.ExposedMonoBehaviour.OnExpose().

Parameters

Event	The name of the event to be invoked.
args	(Optional) Parameters for the event handler.

bool uLua.API.LoadPackage (string *PackageName*) [inline]

Public function used to load a ScriptPackage on demand.

Parameters

PackageName	The name of the ScriptPackage to be loaded.

static void uLua.API.LoadSavedData (LuaMonoBehaviour *Object*, string *Index* = "SaveData") [inline], [static]

Loads table data saved for the specified object.

Saved data will be loaded from the `SaveDataPath` under the external directory. This method is exposed to Lua as a global and may be called from a Lua script or within C#.

Parameters

Object	The object of which the saved data will be loaded.
Index	(Optional) The index of `Object` which contains the table data to be saved. Default value is "SaveData".

static void uLua.API.Log (string *String*) [inline], [static]

Logs a message to the console.

Invokes the event 'LuaMessageLogged' which may be handled in Lua. This method is exposed to Lua as a global and may be called from a Lua script or within C#.

static void uLua.API.LogError (string *String*) [inline], [static]

Logs an error to the console.

Invokes the event 'LuaErrorLogged' which may be handled in Lua. This method is exposed to Lua as a global and may be called from a Lua script or within C#.

static void uLua.API.LogWarning (string *String*) [inline], [static]

Logs a warning to the console.

Invokes the event 'LuaWarningLogged' which may be handled in Lua. This method is exposed to Lua as a global and may be called from a Lua script or within C#.

static void uLua.API.RegisterEvent (string *Event*, string *HandlerName* = "")[inline], [static]

Registers an event to the API.

Events should registered AFTER Unity's Awake call to prevent unexpected behaviour (e.g. on Unity's scene loaded or start events). Events may also be registered when an object is exposed to Lua by overriding uLua.ExposedMonoBehaviour.OnExpose().

Parameters

Event	The name of the event to be registered.
HandlerName	(Optional) The name of the event handler function. Defaults to 'On{Event}', e.g. the default even thandler for an event named 'LoadingFinished' would be 'OnLoadingFinished',

static bool uLua.API.RegisterEventHandler (string *Event*, string *HandlerName*, ILuaObject *Context* = null)[inline], [static]

Registers an event handler to be called when an event is invoked.

This method is exposed to Lua as a global and may be called from a Lua script or within C#.

Parameters

Event	The name of the event.
HandlerName	The name of the event handler function.
Context	(Optional) The object that contains the event handler implementation. If not specified, the event handler will be global.

static void uLua.API.RegisterIndexedType (Type *Type*)[inline], [static]

Registers a type that implements the ILuaObject interface to Lua.

Intended for use with LuaMonoBehaviour and LuaClass.

static void uLua.API.RegisterType< T > ()[inline], [static]

Registers a type to Lua.

Any C# or Unity type may be registered to use within the API. Use with caution.

static void uLua.API.RemoveEventHandlers (LuaMonoBehaviour *Context*)[inline], [static]

Removes all event handlers registered in a specific context.

This method is exposed to Lua as a global and may be called from a Lua script or within C#.

Parameters

Context	The object that contains the event handler implementation.

static void uLua.API.SaveData (LuaMonoBehaviour *Object*, string *Index* = "SaveData")[inline], [static]

Saves table data from the specified object.

Data will be saved to the `SaveDataPath` under the external directory. This method is exposed to Lua as a global and may be called from a Lua script or within C#.

Parameters

Object	The object for which the data will be saved.
Index	(Optional) The index of `Object` which contains the table data to be saved. Default value is "SaveData".

static bool uLua.API.TryRegisterEventHandler (string *HandlerName*, ILuaObject *Context* = `null`)`[inline]`, `[static]`

Attempts to register an event handler for a registered event.

Parameters

HandlerName	The name of the event handler function. If a matching registered event is not found, this function has no effect.
(Optional)	Context The object that contains the event handler implementation. If not specified, the event handler will be global.

uLua.Events Class Reference

Static container class for registered events.

Detailed Description

Static container class for registered events.

uLua.ExposedClass Class Reference

Class structure exposed to Lua. You should use this class as a base for your API data structures.

Public Member Functions

- ExposedClass (string Name, LuaMonoBehaviour Context=null, bool ExposeOnInit=true, bool EnableObjectScript=false)
  Public constructor. Exposes this object to Lua.

Additional Inherited Members

Detailed Description

Class structure exposed to Lua. You should use this class as a base for your API data structures.

Instances of this class are automatically exposed to Lua. All public members of derived classes will also be exposed to Lua. Inherits LuaClass.

Constructor & Destructor Documentation

uLua.ExposedClass.ExposedClass (string *Name*, LuaMonoBehaviour *Context* = `null`, bool *ExposeOnInit* = `true`, bool *EnableObjectScript* = `false`)`[inline]`

Public constructor. Exposes this object to Lua.

If the object is exposed here, this method will also invoke an OnLoad callback.

Parameters

Name	Sets the name of the object exposed to Lua.
Context	Sets the context of the object exposed to Lua.
ExposeOnInit	(Optional) Enables/disables the automatic exposure of this object to Lua.
EnableObjectScript	(Optional) If set to true, executes a resource script after exposing the object to Lua. Resource scripts are only executed for objects which are globals in Lua (i.e. Context is set to null).

uLua.ExposedMonoBehaviour Class Reference

MonoBehaviour script exposed to Lua. You should use this class as a base for your API game objects.

Public Attributes

- ExposeOn **ExposeOn** = ExposeOn.Start
  Indicates when this object will be exposed to Lua.

- bool **EnableObjectScript** = false
  Indicates whether to execute a resource script for this object after it is exposed.

Protected Member Functions

- virtual void Awake ()
  Exposes the game object to Lua if ExposeOn is set to 'Awake'.

- virtual void OnExpose ()
  Scriptable method called when an object is exposed to Lua.

- virtual void Start ()
  Exposes the game object to Lua if ExposeOn is set to 'Start'.

Additional Inherited Members

Detailed Description

MonoBehaviour script exposed to Lua. You should use this class as a base for your API game objects.

Instances of this class are automatically exposed to Lua. All public members of derived classes will also be exposed to Lua. Inherits LuaMonoBehaviour.

Member Function Documentation

virtual void uLua.ExposedMonoBehaviour.Awake () `[inline]`, `[protected]`, `[virtual]`

Exposes the game object to Lua if ExposeOn is set to 'Awake'.

The Context object must be set prior to this method being called. Objects will not be exposed to Lua if `ExposeOn` is set to None. If the object is exposed here, this method will also invoke an OnLoad callback and execute a resource script for the object if `EnableObjectScript` is set to true. This method is called by Unity Engine during game object initialisation.

virtual void uLua.ExposedMonoBehaviour.OnExpose () `[inline]`, `[protected]`, `[virtual]`

Scriptable method called when an object is exposed to Lua.

Executes the object Lua script and invokes `OnLoad` by default.

virtual void uLua.ExposedMonoBehaviour.Start () `[inline]`, `[protected]`, `[virtual]`

Exposes the game object to Lua if ExposeOn is set to 'Start'.

The Context object must be set prior to this method being called. Objects will not be exposed to Lua if `ExposeOn` is set to None. If the object is exposed here, this method will also invoke an OnLoad callback and execute a resource script for the object if `EnableObjectScript` is set to true. Resource scripts are only executed for objects which are globals in Lua (i.e. Context is set to null). This method is called by Unity Engine during game object initialisation.

uLua.ILuaObject Interface Reference

Interface used to implement a fully indexed Lua object.

Properties

- DynValue this[DynValue Key] `[get, set]`
  Returns a Lua value indexed by the DynValue Key.

- DynValue this[string Key] `[get, set]`
  Returns a Lua value indexed by the string key.

- LuaMonoBehaviour Context `[get, set]`
  Used to access/set the context of an object.

- string Handle `[get]`
  Used to access the unique Handle of a Lua object.

- bool IsExposed `[get]`
  Used to track if an object has been exposed to Lua.

- string Name `[get]`
  Used to access the name of an object.

Detailed Description

Interface used to implement a fully indexed Lua object.

Defines various properties (Context, Name, Handle, etc.) which are implemented in LuaClass and LuaMonoBehaviour.

Property Documentation

LuaMonoBehaviour uLua.ILuaObject.Context `[get]`, `[set]`

Used to access/set the context of an object.

The `Context` object represents the parent of an object in Lua. If the `Context` is null, the object will be global in Lua. Must be of type uLua.LuaMonoBehaviour.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

string uLua.ILuaObject.Handle `[get]`

Used to access the unique Handle of a Lua object.

The unique `Handle` of a Lua object is defined as: `Context.Name` . If an object's context is null, its unique handle is simply its name.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

bool uLua.ILuaObject.IsExposed `[get]`

Used to track if an object has been exposed to Lua.

Objects may be marked as exposed by calling the `Expose()` method.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

string uLua.ILuaObject.Name `[get]`

Used to access the name of an object.

The name of an object cannot be changed after it has been exposed to Lua.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

DynValue uLua.ILuaObject.this[DynValue Key] `[get]`, `[set]`

Returns a Lua value indexed by the DynValue Key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

DynValue uLua.ILuaObject.this[string Key] `[get]`, `[set]`

Returns a Lua value indexed by the string key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implemented in uLua.LuaClass, and uLua.LuaMonoBehaviour.

uLua.IndexedUserDataDescriptor Class Reference

Custom user data descriptor to more accurately implement Lua syntax in the Moonsharp interpreter. Makes use of the ILuaObject interface.

Detailed Description

Custom user data descriptor to more accurately implement Lua syntax in the Moonsharp interpreter. Makes use of the ILuaObject interface.

Allows objects in Lua to be indexed with both the '.' and '[]' operators. Credit to user tw39124 from the MoonSharp forum for the original implementation.

uLua.Lua Class Reference

Wrapper class which streamlines use of the MoonSharp Lua context.

Static Public Member Functions

- static void Call (DynValue Function, params object[] args)
  Calls a Lua function with optional parameters.

- static void **Clear** ()
  Resets and initialises the Lua context

- static void ExecuteScript (string Code, string codeFriendlyName="", bool overwriteExisting=false)
  Executes the specified code within the Lua context.

- static void Log (string String)
  Logs a message to the console.

- static void LogError (string String)
  Logs an error to the console.

- static void LogWarning (string String)
  Logs a warning to the console.

- static T Get< T > (string Name)
  Finds and returns the Lua global with the specified name.

- static void ObjectCall< T > (T Object, string FunctionName, params object[] args)
  Calls a Lua object's function.

- static void Remove (string Name)
  Removes a global from the Lua context.

- static void Set (string Index, object Value)
  Exposes an object as a global with the specified index in Lua.

- static DynValue ValueToLuaValue (object Value)
  Converts an object to a DynValue.

Properties

- static DynValue NewFunction `[get]`
  Returns a new Lua function.

- static Table NewTable `[get]`
  Returns a new Lua table.

- static ScriptLoaderBase ScriptLoader `[get, set]`
  Gets or sets the Lua script loader.

Detailed Description

Wrapper class which streamlines use of the MoonSharp Lua context.

Implemented as a static class to ensure the Lua context is available application-wide.

Member Function Documentation

static void uLua.Lua.Call (DynValue *Function*, params object[] *args*) `[inline]`, `[static]`

Calls a Lua function with optional parameters.

Parameters

Function	The `DynValue` of a Lua function.
args	(Optional) Parameters for the Lua function.

static void uLua.Lua.ExecuteScript (string *Code*, string *codeFriendlyName* = `""`, bool *overwriteExisting* = `false`) `[inline]`, `[static]`

Executes the specified code within the Lua context.

Parameters

Code	The Lua code to execute.
codeFriendlyName	(Optional) A unique identifier for this script. Not recommended to leave blank, especially if the same script is likely to be executed more than once.
overwriteExisting	(Optional) If true, the method will overwrite any previous script with the same filename. Otherwise, any previous script will be called from a hash table.

static T uLua.Lua.Get< T > (string *Name*) `[inline]`, `[static]`

Finds and returns the Lua global with the specified name.

The returned object is converted to the type specified by the generic type parameter `T` .

static void uLua.Lua.Log (string *String*) `[inline]`, `[static]`

Logs a message to the console.

Invokes the event 'LuaMessageLogged' which may be handled in Lua. This method is exposed to Lua as a global and may be called from a Lua script or within C#.

static void uLua.Lua.LogError (string *String*) `[inline]`, `[static]`

Logs an error to the console.

Invokes the event 'LuaErrorLogged' which may be handled in Lua. This method is exposed to Lua as a global and may be called from a Lua script or within C#.

static void uLua.Lua.LogWarning (string *String*) `[inline]`, `[static]`

Logs a warning to the console.

Invokes the event 'LuaWarningLogged' which may be handled in Lua. This method is exposed to Lua as a global and may be called from a Lua script or within C#.

static void uLua.Lua.ObjectCall< T > (T *Object*, string *FunctionName*, params object[] *args*) `[inline]`, `[static]`

Calls a Lua object's function.

Parameters

Object	The object whose function will be called. The generic type `T` must implement the `ILuaObject` interface.
FunctionName	The name of the callback function to be called.
args	(Optional) Parameters for the callback function.

Type Constraints

T : *ILuaObject*

static void uLua.Lua.Remove (string *Name*) `[inline]`, `[static]`

Removes a global from the Lua context.

Any Lua objects defined as members of the global are invalidated.

static void uLua.Lua.Set (string *Index*, object *Value*) `[inline]`, `[static]`

Exposes an object as a global with the specified index in Lua.

For object types that implement the ILuaObject interface, use `uLua.API.Expose()` instead.

Parameters

Index	The index, or name, of the value within Lua.
Value	The value to be defined as a global within Lua.

static DynValue uLua.Lua.ValueToLuaValue (object *Value*) `[inline]`, `[static]`

Converts an object to a DynValue.

May be used with any C# object type.

Property Documentation

DynValue uLua.Lua.NewFunction `[static]`, `[get]`

Returns a new Lua function.

Used as a placeholder for empty callback functions.

Table uLua.Lua.NewTable `[static]`, `[get]`

Returns a new Lua table.

Creates a new Table type Lua value.

ScriptLoaderBase uLua.Lua.ScriptLoader `[static]`, `[get]`, `[set]`

Gets or sets the Lua script loader.

The script loader is set on awake in the API script.

uLua.LuaClass Class Reference

Class structure which establishes a Lua object framework. For internal use.

Public Member Functions

- LuaClass (string Name, LuaMonoBehaviour Context=null)
  Public constructor.

- void Expose ()
  Used to raise a flag when an object is exposed to Lua.

- void InvokeLua (string FunctionName, params object[] args)
  Invokes a Lua callback function.

- void Register (string FunctionName, string Code="", string args="")
  Registers a Lua callback function. This method is available in Lua.

Properties

- DynValue this[DynValue Key] `[get, set]`
  Returns a Lua value indexed by the DynValue Key.

- DynValue this[string Key] `[get, set]`
  Returns a Lua value indexed by the string key.

- LuaMonoBehaviour Context `[get, set]`
  Used to access/set the context of an object.

- string Handle `[get]`
  Used to access the unique Handle of a Lua object.

- bool IsExposed `[get]`
  Used to track if an object has been exposed to Lua.

- string Name `[get, protected set]`
  Used to access the name of an object.

Detailed Description

Class structure which establishes a Lua object framework. For internal use.

Instances of this class can be exposed to Lua by calling `uLua.API.Expose()` . All public members of derived classes will be exposed to Lua. Prior to exposing an object of type uLua.LuaClass, the derived type `T` must be registered to Lua by calling `uLua.API.RegisterIndexedType()` .

Constructor & Destructor Documentation

uLua.LuaClass.LuaClass (string *Name*, LuaMonoBehaviour *Context* = `null`) `[inline]`

Public constructor.

Parameters

Name	Sets the name of the object exposed to Lua.
Context	Sets the context of the object exposed to Lua.

Member Function Documentation

void uLua.LuaClass.Expose () `[inline]`

Used to raise a flag when an object is exposed to Lua.

The property `IsExposed` can be used to check if this method has been called. The `IsExposed` flag cannot be reset.

Implements uLua.ILuaObject.

void uLua.LuaClass.InvokeLua (string *FunctionName*, params object[] *args*) `[inline]`

Invokes a Lua callback function.

The callback function must be implemented in a Lua script as a member of this object. Alternatively, you can register a callback function by explicitly calling uLua.LuaMonoBehaviour.Register() with the relevant Lua code.

Parameters

FunctionName	The name of the callback function to be called.
args	(Optional) Parameters for the callback function.

void uLua.LuaClass.Register (string *FunctionName*, string *Code* = `""`, string *args* = `""`) `[inline]`

Registers a Lua callback function. This method is available in Lua.

Used to register a callback function by providing the relevant Lua code.

Parameters

FunctionName	The name of the callback function to be registered.
Code	(Optional) The body of the function to be registered in Lua. If omitted, an empty function with the specified name is defined.
args	(Optional) Parameters for the callback function. Must be separated by comma as they would be in a Lua function definition, e.g. 'arg1,arg2'.

Property Documentation

LuaMonoBehaviour uLua.LuaClass.Context `[get]`, `[set]`

Used to access/set the context of an object.

The `Context` object represents the parent of an object in Lua. If the `Context` is null, the object will be global in Lua. Must be of type uLua.LuaMonoBehaviour.

Implements uLua.ILuaObject.

string uLua.LuaClass.Handle `[get]`

Used to access the unique Handle of a Lua object.

The unique `Handle` of a Lua object is defined as: `Context.Name` . If an object's context is null, its unique handle is simply its name.

Implements uLua.ILuaObject.

bool uLua.LuaClass.IsExposed `[get]`

Used to track if an object has been exposed to Lua.

Objects may be marked as exposed by calling the `Expose()` method.

Implements uLua.ILuaObject.

string uLua.LuaClass.Name `[get]`, `[protected set]`

Used to access the name of an object.

The name of an object cannot be changed after it has been exposed to Lua.

Implements uLua.ILuaObject.

DynValue uLua.LuaClass.this[DynValue Key] `[get]`, `[set]`

Returns a Lua value indexed by the DynValue Key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implements uLua.ILuaObject.

DynValue uLua.LuaClass.this[string Key] `[get]`, `[set]`

Returns a Lua value indexed by the string key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implements uLua.ILuaObject.

uLua.LuaMonoBehaviour Class Reference

MonoBehaviour script which establishes a Lua object framework. For internal use.

Public Member Functions

- void Expose ()
  Used to raise a flag when an object is exposed to Lua.
- void InvokeLua (string FunctionName, params object[] args)
  Invokes a Lua callback function.
- void Register (string FunctionName, string Code="", string args="")
  Registers a Lua callback function. This method is available in Lua.

Protected Member Functions

- virtual void OnDestroy ()
  Invokes the OnExit callback and removes the object from Lua.

Properties

- DynValue this[DynValue Key] `[get, set]`
  Returns a Lua value indexed by the DynValue Key.
- DynValue this[string Key] `[get, set]`
  Returns a Lua value indexed by the string key.
- LuaMonoBehaviour Context `[get, set]`
  Used to access/set the context of an object.
- string Handle `[get]`
  Used to access the unique Handle of a Lua object.
- bool IsExposed `[get]`
  Used to track if an object has been exposed to Lua.
- string Name `[get, set]`
  Used to access/set the name of an object.

Detailed Description

MonoBehaviour script which establishes a Lua object framework. For internal use.

Instances of this class can be exposed to Lua by calling `uLua.API.Expose()` . All public members of derived classes will be exposed to Lua. Prior to exposing an object of type uLua.LuaMonoBehaviour, the derived type `T` must be registered to Lua by calling `API.RegisterIndexedType()`.

Member Function Documentation

void uLua.LuaMonoBehaviour.Expose () `[inline]`

Used to raise a flag when an object is exposed to Lua.

The property `IsExposed` can be used to check if this method has been called. The `IsExposed` flag cannot be reset.

Implements uLua.ILuaObject.

void uLua.LuaMonoBehaviour.InvokeLua (string *FunctionName*, params object[] *args*) `[inline]`

Invokes a Lua callback function.

The callback function must be implemented in a Lua script as a member of this object. Alternatively, you can register a callback function by explicitly calling uLua.LuaMonoBehaviour.Register() with the relevant Lua code.

Parameters

FunctionName	The name of the callback function to be called.
args	(Optional) Parameters for the callback function.

virtual void uLua.LuaMonoBehaviour.OnDestroy () `[inline]`, `[protected]`, `[virtual]`

Invokes the OnExit callback and removes the object from Lua.

This method is called by Unity Engine when a game object is destroyed.

void uLua.LuaMonoBehaviour.Register (string *FunctionName*, string *Code* = `""`, string *args* = `""`) `[inline]`

Registers a Lua callback function. This method is available in Lua.

Used to register a callback function by providing the relevant Lua code.

Parameters

FunctionName	The name of the callback function to be registered.
Code	(Optional) The body of the function to be registered in Lua. If omitted, an empty function with the specified name is defined.
args	(Optional) Parameters for the callback function. Must be separated by comma as they would be in a Lua function definition, e.g. 'arg1,arg2'.

Property Documentation

LuaMonoBehaviour uLua.LuaMonoBehaviour.Context `[get]`, `[set]`

Used to access/set the context of an object.

The `Context` object represents the parent of an object in Lua. If the `Context` is null, the object will be global in Lua. Must be of type uLua.LuaMonoBehaviour.

Implements uLua.ILuaObject.

string uLua.LuaMonoBehaviour.Handle `[get]`

Used to access the unique Handle of a Lua object.

The unique `Handle` of a Lua object is defined as: `Context.Name` . If an object's context is null, its unique handle is simply its name.

Implements uLua.ILuaObject.

bool uLua.LuaMonoBehaviour.IsExposed `[get]`

Used to track if an object has been exposed to Lua.

Objects may be marked as exposed by calling the `Expose()` method.

Implements uLua.ILuaObject.

string uLua.LuaMonoBehaviour.Name `[get]`, `[set]`

Used to access/set the name of an object.

If a name is not set, the game object's name will be used by default. The name of an object cannot be changed after it has been exposed to Lua.

Implements uLua.ILuaObject.

DynValue uLua.LuaMonoBehaviour.this[DynValue Key] `[get]`, `[set]`

Returns a Lua value indexed by the DynValue Key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implements uLua.ILuaObject.

DynValue uLua.LuaMonoBehaviour.this[string Key] `[get]`, `[set]`

Returns a Lua value indexed by the string key.

This indexer may be used in Lua or within C# to access any member of a Lua object.

Implements uLua.ILuaObject.

uLua.ScriptLoader Class Reference

A script loader to load scripts from assets in Unity from within the Lua context. Scripts will be loaded from a subdirectory of Assets/Resources.

Public Member Functions

- ScriptLoader (string AssetsPath)
  Initializes a new instance of the uLua.ScriptLoader class.

- override object LoadFile (string file, Table globalContext)
  Opens a file for reading the script code. It can return either a string, a byte[] or a Stream. If a byte[] is returned, the content is assumed to be a serialized (dumped) bytecode. If it's a string, it's assumed to be either a script or the output of a string.dump call. If a Stream, autodetection takes place.

- override bool ScriptFileExists (string file)
  Checks if a given file exists.

Detailed Description

A script loader to load scripts from assets in Unity from within the Lua context. Scripts will be loaded from a subdirectory of Assets/Resources.

This class is used by the loadfile and require functions in Lua.

Constructor & Destructor Documentation

uLua.ScriptLoader.ScriptLoader (string *AssetsPath*)`[inline]`

Initializes a new instance of the uLua.ScriptLoader class.

Parameters

AssetsPath	The path, relative to Assets/Resources. For example if your scripts are stored under Assets/Resources/Scripts/, you should pass the value "Scripts/".

Member Function Documentation

override object uLua.ScriptLoader.LoadFile (string *file*, Table *globalContext*)`[inline]`

Opens a file for reading the script code. It can return either a string, a byte[] or a Stream. If a byte[] is returned, the content is assumed to be a serialized (dumped) bytecode. If it's a string, it's assumed to be either a script or the output of a string.dump call. If a Stream, autodetection takes place.

Parameters

file	The file.
globalContext	The global context.

Returns

A string, a byte[] or a Stream.

Exceptions

System.Exception	UnityAssetsScriptLoader.LoadFile : Cannot load + file

override bool uLua.ScriptLoader.ScriptFileExists (string *file*)`[inline]`

Checks if a given file exists.

Parameters

file	The file.

Returns

True if the file exists, false if it does not.

uLua.ScriptPackage Class Reference

A class to describe Lua packages.

Public Member Functions

- ScriptPackage (string Name, string FolderName, ScriptPackageJson PackageJson, bool IsExternal)
  Public constructor. This is the main way to initialise a ScriptPackage object.

- void **SwitchToExternal** ()
  This method is used to switch the package definition to external. Using this method instead of a public property protects the accessibility of IsExternal.

Properties

- bool **AllowExternalOverride** `[get]`
  Determines whether a ScriptPackage defined in resources can be overriden by an external definition.

- bool **LoadAllFiles** `[get]`
  Determines which scripts of a ScriptPackage will be executed. If true, all scripts found under the ScriptPackage folder will be executed. Otherwise, only scripts specified in the Contents list will be executed.

- bool **LoadOnDemand** `[get]`
  Determines whether the ScriptPackage will be loaded on demand or on load. If true, the ScriptPackage will not be loaded until explicitly requested. Otherwise, it will be loaded during on scene initialisation.

- string[] **Contents** `[get]`
  Gets the array of script names specified as contents of the ScriptPackage.

- string[] **Dependencies** `[get]`
  Gets the array of dependencies specified for the ScriptPackage.

- string **Description** `[get]`
  Gets the description of the ScriptPackage.

- string **FolderName** `[get]`
  Gets the name of the folder in which the ScriptPackage is contained (i.e. if it differs from the index name of the Lua package).

- bool **IsExternal** `[get]`
  Gets the IsExternal property of the package.

- string **Name** `[get]`
  Gets the index name of the ScriptPackage.

- string **Title** `[get]`
  Gets the title of the ScriptPackage for display purposes.

- string **Version** `[get]`
  Gets the version string of the ScriptPackage.

Detailed Description

A class to describe Lua packages.

Constructor & Destructor Documentation

uLua.ScriptPackage.ScriptPackage (string *Name*, string *FolderName*, ScriptPackageJson *PackageJson*, bool *IsExternal*) `[inline]`

Public constructor. This is the main way to initialise a ScriptPackage object.

Parameters

Name	The name of the ScriptPackage.
FolderName	The name of the folder in which the ScriptPackage is contained (i.e. if it differs from the name of the Lua package).
PackageJson	The object which holds most information for the Lua package.
IsExternal	Specifies if the script package was defined as an external package.

uLua.ScriptPackageJson Class Reference

A simple structure used to load ScriptPackage information from a json file using Unity's JsonUtility.

Detailed Description

A simple structure used to load ScriptPackage information from a json file using Unity's JsonUtility.

Index