uLua

A N T

Software

uLua

ANT Software

Version v1.3.0

Sat Feb 11 2023

Table of Contents

uLua

uLua is a Lua Modding framework for Unity. It enables the development of a Lua API which may be used to add modding functionality to your Unity Project.

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.
- The ability to organise user scripts in packages which can be easily installed and removed.
- Event and callback system which allows events and functions to be invoked in C# but implemented in Lua.
- Base classes to expose your game objects and data structures to Lua.
- Ability to override C# methods in Lua for specific objects.

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!

Dependencies

- MoonSharp for Unity

Resources

Below are some projects in which I have made use of uLua in Unity Engine. These are by no means complete products, but can be used as a demo use of this toolkit.

- **Demo Game:** A demo game which uses uLua, and comes with its own documentation of the API and source code.
- **uLua.Essentials:** A set of API objects which implement structures for an application manager, keybindings, an in-game console, and a profile manager.
- **uLua.Objects:** A set of abstract classes which implement critical features of Unity Engine game objects in a Lua API, and an accompanying object manager class.

Documentation

This is the complete documentation for uLua. For any further questions do not hesitate to contact support@antsoftware.co.uk.

Usage Tutorial

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

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 objects 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.

The following settings can be configured in the inspector UI:

- **ResourcePath** : Path for Lua scripts within the Resources folder.
- **ExternalDirectory** : Directory for external Lua scripts. `Application.persistentDataPath` is used unless otherwise specified.
- **SaveDataPath** : Path for user save data under `ExternalDirectory` .
- **UserScriptsPath** : Path for user scripts under `ExternalDirectory` . Scripts in this path are loaded automatically and executed if `EnableUserScripts` is enabled.
- **ScriptPackagesPath** : Path for script packages under `ExternalDirectory` .
- **LuaScriptsExtension** : The file extension for all external Lua scripts. Set to *lua* by default.
- **EnableSceneScript** : If enabled, the uLua.API class attempts to load a script by the name of the scene. The scene script may be located in the `ResourcePath` or in the `UserScriptsPath` .
- **EnableUserScripts** : If enabled, allows user scripts to be executed which extend the `SceneScript` .
- **EnableConsoleMessages** : If enabled, messages will be output to the console at runtime.

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

For now, you can stick to the default settings and continue to the next section.

2. Executing Lua scripts

The uLua.API class provides a scripting framework which allows you to execute Lua code by wrapping around the MoonSharp Lua interpreter. Lua scripts may be executed from the Resources folder of your project or from an external directory.

In the following sections we will go through different approaches to executing Lua scripts.

2.1. Loading a Scene Script

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

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 `ResourcePath` under the Resources folder in your project. The `ResourcePath` 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 notice the message "Hello World!" output to the console. Congratulations, you have executed your first Lua script!

2.2. Loading a Scene Script Externally

The uLua.API class also allows you to load the scene script from an external directory. To do so, you must use the `EnableUserScripts` option of the uLua.API class from the inspector UI. You must then move your scene script to the external directory under the designated `UserScriptsPath` . When loaded externally, 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}/{UserScriptsPath}/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 `UserScriptsPath` is set to "Scripts" by default.

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

When the `EnableSceneScript` option is 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. The intended use of this feature is to allow users to override the default implementation of scene scripts by redefining them at runtime. This is very useful if you want to allow users to modify your game behaviour.

2.3. User Scripts

In the 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 user scripts which are executed after the scene script. To enable user scripts, use the `EnableUserScripts` option in the inspector UI.

User 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 user scripts:

```
{ExternalDirectory}/{UserScriptsPath}/
{ExternalDirectory}/{UserScriptsPath}/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 `UserScriptsPath` 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 a user script.");
```

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

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

2.4. Script Packages

Script packages is a feature which allows users to organise scripts in groups (from here on called packages). Organising scripts in packages provides more control of the order of execution of different scripts, as well as easy installation and removal of several scripts at once.

2.4.1. Script package structure

Script packages are parsed and loaded **after** the base user scripts described in the previous section. Scripts must be placed in a folder under the designated `ScriptPackagesPath`. For instance, a package which is named `MyPackage` must be located in the following directory:

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

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

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 that the API class ignores all characters prior to the first dash in the folder name when parsing for the name of the script package.

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"
}
```

The properties listed in this example are the title, description, and version of the package which can be used for display purposes. In the following sections, we cover how to define the contents and dependencies of a package, and how to access installed packages.

2.4.2. Package contents and order of execution

Some properties defined in the package json file allow 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 during on scene initialisation. `LoadOnDemand` defaults to false if omitted.

2.4.3. Dependencies

If 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.

2.4.4. Package API Functions

The data defined in the package json file is processed by the API class during the execution of the package. In addition, 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.

2.5. Manual Script Execution

So far we have used various built-in settings of the uLua.API class to execute Lua scripts. In case the implemented functionality does not fit your project workflow, the uLua.API class also allows you to execute Lua scripts on demand from anywhere in your project.

The following two static methods are available to use in your code:

uLua.API.ExecuteExternalFile()

uLua.API.ExecuteFile()

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.

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");
```

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 Event Handlers

Event handlers are registered and removed by the uLua.API class. The related methods are static, which means they can be executed from anywhere in your project. Until an event handler has been registered, the uLua.API.Invoke() command will have no effect.

To register an event handler for the `PlayerHealthChanged` event you can use the following command in C#:

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

This command registers the `OnPlayerHealthChanged` global Lua function to be called when the `PlayerHealthChanged` event is invoked. The event handler `OnPlayerHealthChanged` must be defined ***before*** the uLua.API.RegisterEventHandler() command is called, otherwise invoking the `PlayerHealthChanged` event will cause a runtime exception.

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 `OnPlayerHealthChanged` function.

MyGameScene.lua

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

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
```

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 OnPlayerHealthChanged()
    -- Do stuff
end

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

These two approaches are equivalent in functionality. However, allowing event handlers to be registered by the user in Lua scripts allows for a lot more flexibility in script design.

3.3. Removing Event Handlers

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.

3.4. 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 user scripts are executed. The `SceneUnloaded` event is invoked when a scene is unloaded. You may register 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 OnPlayerHealthChanged()
    -- Do stuff
end

function OnSceneLoaded()
    -- Do other stuff
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
RegisterEventHandler("SceneUnloaded", "OnSceneUnloaded");
```

4. Exposing Objects to Lua

In the previous two sections we have shown how the scene script 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 exposing itself to Lua.

When a class inherits uLua.ExposedMonoBehaviour, all its public members are accessible in Lua. This makes uLua.ExposedMonoBehaviour the 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 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, all game objects which contain the `Player` script will be exposed to the API when the object's `Start()` method is called. This option may be changed by setting the `ExposeOn` parameter of uLua.ExposedMonoBehaviour to `Awake` , or to `None` , allowing you to expose objects manually by using uLua.API.Expose<T>().

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 OnPlayerHealthChanged()
    -- Do stuff
end

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

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
RegisterEventHandler("SceneUnloaded", "OnSceneUnloaded");
```

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 have made the `Health` variable private and renamed

it to `_Health` . We then implement a public getter property named `Health` , and 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 {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

        API.Invoke("PlayerHealthChanged");
    }

    public int Health {
        get { return _Health; }
    }
}
```

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

MyGameScene.lua

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

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(5);
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
RegisterEventHandler("SceneUnloaded", "OnSceneUnloaded");
```

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 game objects. For instance, if your Player object is comprised of several components (as is common in Unity), and you need properties of these components to be accessible in Lua, then you will 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 EnableResourceScript = false): base(Name, Context, ExposeOnInit,
EnableResourceScript) {
    }
}
```

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 `EnableResourceScript` 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 `Awake()` method of the `Player` script.

Player.cs

```
using uLua;

public class Player : ExposedMonoBehaviour {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

        API.Invoke("PlayerHealthChanged");
    }

    public int Health {
        get { return _Health; }
    }

    private void Awake() {
        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 OnPlayerHealthChanged()
    print (MyPlayer.Health);
end

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(Sword.Damage);
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
RegisterEventHandler("SceneUnloaded", "OnSceneUnloaded");
```

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 feature 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. 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 {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

        API.Invoke("PlayerHealthChanged");
    }

    public int Health {
        get { return _Health; }
    }

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

In this case we have instantiated a second instance of `Weapon`. Both instances are named `Sword`. However, the `PlayerSword` instance is in the context of the `Player` object. 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

```
Sword.Damage = 5;
MyPlayer.Sword.Damage = 2;
```

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

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(Sword.Damage);
end

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
RegisterEventHandler("SceneUnloaded", "OnSceneUnloaded");
```

Here we have added a command to set the `Sword.Damage` property to 2 for the `Sword` object which is in the context of our `MyPlayer` object.

The `Context` is 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 {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

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

    public int Health {
        get { return _Health; }
    }

    public void Awake() {
        Weapon Sword = new Weapon("Sword");
        Weapon PlayerSword = 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

```
Sword.Damage = 5;
MyPlayer.Sword.Damage = 2;

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

function OnSceneLoaded()
    print (MyPlayer.Health);

    MyPlayer:Damage(Sword.Damage);
end

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

function OnSceneUnloaded()
    RemoveEventHandlers();
end

RegisterEventHandler("PlayerHealthChanged", "OnPlayerHealthChanged");
RegisterEventHandler("SceneLoaded", "OnSceneLoaded");
RegisterEventHandler("SceneUnloaded", "OnSceneUnloaded");
```

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 user scripts.

- To use this feature for instances of uLua.ExposedMonoBehaviour, you must enable the `EnableResourceScript` option in the inspector UI of a certain object.
- To use this feature for instances of uLua.ExposedClass, you must use the `EnableResourceScript` 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 `ResourcePath` 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 `UserScriptsPath` 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 {
    private int _Health = 100;

    [AllowLuaOverride]
    public void Damage(int Damage) {
        _Health -= Damage;

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

    public int Health {
        get { return _Health; }
    }

    public void Awake() {
        Weapon Sword = new Weapon("Sword");
        Weapon PlayerSword = 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 User Scripts

It is often useful to hide certain objects of the API from users. To achieve this, you may use the `UserBlacklist` feature. This is available in the API class through the Unity

inspector. Adding the name of a global object to the `UserBlacklist` list will make it unavailable to user 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("Awake")]
public class Player : ExposedMonoBehaviour {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

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

    public int Health {
        get { return _Health; }
    }

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

This would prevent the `Awake()` 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 resource script, or
- in user 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 {
    private int _Health = 100;

    public void Damage(int Damage) {
        _Health -= Damage;

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

    public int Health {
        get { return _Health; }
    }

    public void Awake() {
        Weapon Sword = new Weapon("Sword");
        Weapon PlayerSword = 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` would be 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

RegisterEventHandler("LuaMesageLogged", "OnLuaMessageLogged");

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

7. 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 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 IHasLuaIndexer
  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 IHasLuaIndexer 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 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

- void AddUserScript (string Name)
  Adds a script to the user scripts by name.

- void ClearUserScripts ()
  Clears the user scripts list.

- 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.

- void RemoveUserScript (string Name)
  Removes the specified script from the user scripts.

Static Public Member Functions

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

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

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

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

- static void RegisterIndexedType< T > ()
  Registers a type that implements the IHasLuaIndexer interface to Lua.

- static void RegisterIndexedType (Type Type)
  Registers a type that implements the IHasLuaIndexer 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 Public Attributes

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

- static string **ResourcePath** = ""
  Path for Lua scripts within the Resources folder.

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

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

- 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

void uLua.API.AddUserScript (string *Name*) [inline]

Adds a script to the user scripts by name.

User scripts are executed from the specified path in the `ExternalDirectory` after the scene is loaded.

void uLua.API.ClearUserScripts () [inline]

Clears the user scripts list.

This method has no effect on the API if the user scripts have already been executed.

static bool uLua.API.ExecuteExternalFile (string *File*, string *Index* = "", bool *Force* = false) [inline], [static]

Executes a script from the external directory.

Parameters

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

static bool uLua.API.ExecuteFile (string *File*, string *Index* = "", bool *Force* = false) [inline], [static]

Executes a script from the resource directory.

Parameters

File	The Lua file to execute. The file extension must be omitted.
Index	(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.
Force	(Optional) If true, the method will overwrite any previous script with the same Index. Otherwise, any previous script with the same Index 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 IHasLuaIndexer interface.
Context	The Lua context of the object. If null, the object will be exposed as a global.

Type Constraints

T : *class*
T : *IHasLuaIndexer*

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#.

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.RegisterEventHandler (string *Event*, string *HandlerName*, LuaMonoBehaviour *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 IHasLuaIndexer interface to Lua.

Intended for use with LuaMonoBehaviour and LuaClass.

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

Registers a type that implements the IHasLuaIndexer interface to Lua.

Intended for use with LuaMonoBehaviour and LuaClass.

Type Constraints

T : *class*
T : *IHasLuaIndexer*

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* = `null`)`[inline], [static]`

Removes all global event handlers or 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	(Optional) The object that contains the event handler implementation. If not specified, all global event handlers will be removed.

void uLua.API.RemoveUserScript (string *Name*)`[inline]`

Removes the specified script from the user scripts.

User scripts are executed from the specified path in the `ExternalDirectory` after the scene is loaded.

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".

uLua.ExposedClass< T > Class Template 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 EnableResourceScript=false)
  Public constructor. Exposes this object to Lua.

- ExposedClass (string Name, LuaMonoBehaviour Context=null, bool ExposeOnInit=true, bool EnableResourceScript=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.

Instances of this class are automatically exposed to Lua. The generic type parameter `T` is used to register that type to Lua. All public members of derived classes will be exposed to Lua. Inherits LuaClass.

Type Constraints

T : *class*
T : *IHasLuaIndexer*

Constructor & Destructor Documentation

uLua.ExposedClass< T >.ExposedClass (string *Name*, LuaMonoBehaviour *Context* = `null`, bool *ExposeOnInit* = `true`, bool *EnableResourceScript* = `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.
EnableResourceScript	(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.ExposedClass< T >.ExposedClass (string *Name*, LuaMonoBehaviour *Context* = `null`, bool *ExposeOnInit* = `true`, bool *EnableResourceScript* = `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.
EnableResourceScript	(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< T > Class Template 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 **EnableResourceScript** = 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 Start ()
  Exposes the game object to Lua if ExposeOn is set to 'Start'.

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

- 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.

Instances of this class are automatically exposed to Lua. The generic type parameter `T` is used to register that type to Lua. All public members of derived classes will be exposed to Lua. Inherits LuaMonoBehaviour.

Type Constraints

T : *class*
T : *IHasLuaIndexer*

Member Function Documentation

virtual void uLua.ExposedMonoBehaviour< T >.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 `EnableResourceScript` is set to true. This method is called by Unity Engine during game object initialisation.

virtual void uLua.ExposedMonoBehaviour< T >.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 `EnableResourceScript` is set to true. This method is called by Unity Engine during game object initialisation.

virtual void uLua.ExposedMonoBehaviour< T >.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 `EnableResourceScript` 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.

virtual void uLua.ExposedMonoBehaviour< T >.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 `EnableResourceScript` 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.IHasLuaIndexer 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.IHasLuaIndexer.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.IHasLuaIndexer.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.IHasLuaIndexer.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.IHasLuaIndexer.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.IHasLuaIndexer.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.IHasLuaIndexer.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< T > Class Template Reference

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

Detailed Description

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

Type Constraints

T : *class*
T : *IHasLuaIndexer*

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 ExecuteScript (string Code, string Index="", bool Force=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.

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 *Index* = "", bool *Force* = false) [inline], [static]

Executes the specified code within the Lua context.

Parameters

Code	The Lua code to execute.
Index	(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.
Force	(Optional) If true, the method will overwrite any previous script with the same index. Otherwise, any previous script with the same index 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 `IHasLuaIndexer` interface.
FunctionName	The name of the callback function to be called.
args	(Optional) Parameters for the callback function.

Type Constraints

T : *IHasLuaIndexer*

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 IHasLuaIndexer 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.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

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.IHasLuaIndexer.

uLua.ScriptPackage Class Reference

A class to describe Lua packages.

Public Member Functions

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

Properties

- 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).

- 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*) `[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.

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