Julia Unity Package - Documentation

Important Files and Folders:

StreamingAssets Folder:

- Special folder inside the Assets folder.
- Referenced with `Application.streamingAssetsPath`.
- Create this folder inside your project's Assets folder if absent or import it from the package's samples.
- See also the Unity documentation

Inclusions.txt:

- Text file in Assets/StreamingAssets folder responsible for listing the Julia files needed for your project.
- Add a new file reference by writing `filename.jl` into a new line.
- Move the Inclusions.txt file from Samples~ into your project's StreamingAssets folder.
- Referenced with `FileHandler.Instance.InclusionsFilePath`.
- Commenting on this file can be done by placing a `#` at the start of a new line.
- Make sure that every line either contains a valid Julia file or starts with a `#`.
- The referenced files are loaded with `JuliaModelHandler.LoadModelDependencies()`.
- Template for Inclusions.txt:

```
# include operations for Julia
# place this file in your project's StreamingAssets folder
# used for including Julia files placed in the JuliaScripts folder
# use: filename.jl
# lines cam either contain a valid Julia file (filename.jl) or start with '#'
# lines with '#' at the beginning are not included!
# you can use '#' for commenting the file, just as you can see right now.
load_dependencies.jl
```

JuliaScripts Folder:

- Folder at the root of the current working directory.
- Move the JuliaScripts folder from the Samples~ folder into your project's directory and after building into the standalone's directory.
- Referenced with `Path.Combine(Directory.GetCurrentDirectory(), "JuliaScripts")` .
- Julia files in this folder should not contain the following code since it corrupts the Project's structure:

```
// this code sets the current working directory to the folder this code's
file is located in
cd(dirname(@__DIR__))
```

Folder Structure:

- Unity Project:

```
Unity Project Directory
├── Assets
├── Packages
└── JuliaScripts
```

- Standalone:

```
Standalone Directory
├── Standalone.exe
├── Standalone_Data
└── JuliaScripts
```

load_dependencies.jl:

- Julia file in JuliaScripts folder. Responsible for including Julia packages that are needed by additional Julia scripts.
- Add a new package by adding the following code to the load_dependecies.jl file:

```
try
    using PackageName
catch e
    import Pkg; Pkg.add("PackageName")
    using PackageName
end
```

- replace *PackageName* with the name of the needed Julia package

helper.jl:

- Julia file in JuliaScripts folder.
- Is referenced by the JuliaBase class and handles accessing data and sub-arrays of multi-dimensional arrays and the conversion of matrices into one-dimensional arrays.
- For more see `JuliaBase.GetFromDimN` functions (N ϵ {One, Two, Three, Four}).

Important Classes, Interfaces their Attributes and Functions:

Note: Since markdown isn't able to display square brackets in code entries the **[ ]** are replaced by **{ }**.

JuliaBase:

- The class that handles the communication with Julia and the representation of data from Julia.

Functions:

- `void Init()`
  - Starts Julia and makes Julia's logs available for feedback.
- `void Exit()`
  - Sends Julia the command to finish its process.
- `void InitHelperFunctions()`
  - Loads and references the JuliaFunctions needed for extracting Values and one-dimensional Arrays from multi-dimensional Julia Arrays, and for converting matrices into one-dimensional arrays.
- `JuliaValue EvalString(string message))`
  - Evaluates an instruction for Julia passed as a String. The string has to follow the Julia syntax and be valid code, otherwise Julia will throw an error. EvalString() returns a JuliaValue object, if the instruction returns a value.
- `void EvalStringArray(string { } stringArray)`
  - Evaluates multiple instructions for Julia passed as a String array. The strings have to follow the Julia syntax and be valid code, otherwise Julia will throw an error.
- `JuliaFunction GetJuliaFunctionsByModule(JuliaModule module, string funcName)`
  - Gets a reference to a loaded Julia function that is part of a Julia module.
- `JuliaValue GetFromDimOne(JuliaValue array, int indexDimOne)`
  - Calls the Julia function getFromDimOne and extracts a JuliaValue from the 1st dimension of an n-dimensional array.
- `JuliaValue GetFromDimTwo(JuliaValue array, int indexDimOne, int indexDimTwo)`
  - Calls the Julia Function getFromDimTwo and extracts a JuliaValue from the 2nd dimension of an n-dimensional array.
- `JuliaValue GetFromDimThree(JuliaValue array, int indexDimOne, int indexDimTwo, int indexDimThree)`
  - Calls the Julia Function getFromDimThree and extracts a JuliaValue from the 3rd dimension of an n-dimensional array.
- `JuliaValue GetFromDimFour(JuliaValue array, int indexDimOne, int indexDimTwo, int indexDimThree, int indexDimFour)`
  - Calls the Julia Function getFromDimFour and extracts a JuliaValue from the 4th dimension of an n-dimensional array.
- `JuliaValue MatrixToArray(JuliaValue matrix)`
  - Converts a Julia matrix into a one-dimensional Julia array with the same length. Use `AsArray1D<T>()` to translate the Julia array into a C# array.
- `string BaseInclude(string moduleName, string funcName)`
  - Generates the include() command of Julia's Base module for a specific module and a specific function.
- `string BaseInclude(string funcName)`
  - Generates the include() command of Julia's Base Module for the Main Module and a specific Function.

Structs:

The structs of the JuliaBase class represent various data structures in Julia.

JuliaValue

- `UnboxType`
    - The UnboxType properties translate JuliaValues of the Julia type *Type* into the corresponding C# type. For example UnboxFloat64 translates a value of type Float64 into a Double. Here is a list of all `UnboxType` properties:

```
UnboxBool
UnboxFloat64
UnboxFloat32
UnboxInt64
UnboxInt32
UnboxInt16
```

- `JuliaValue Wrap(object toWrap)`
    - Converts a value of C# type T into its corresponding Julia type. For example a value of type Double will be converted to a JuliaValue referencing a Float64.
- `T{} AsArrayND<T>()` | N ϵ {1, 2, 3}
    - Copies the data of a Julia array into a new one-dimensional C# array of type T. T needs to be the corresponding C# type to the type of the contained values. This doesn't work for Julia arrays with the type Any. If the array has type Any it is recommended to divide it into its contained values and single-type sub-arrays by using the `GetFromDimN()` methods. Here is a list of all `AsArrayND<T>()` methods:

```
T{} AsArray1D<T>()
T{} AsArray2D<T>()
T{} AsArray3D<T>()
```

JuliaFunction

- `JuliaFunction(IntPtr wrapped)`
    - A Constructor that takes an IntPtr.
- `JuliaFunction(JuliaValue juliaValue)`
    - A Constructor that takes a JuliaValue and converts it into a JuliaFunction.
- `JuliaValue Invoke(params JuliaValue{} args)`
    - Calls a referenced Julia function with the given parameters as JuliaValues.
- `JuliaValue Invoke(object first, params object{ } args)`
    - Calls a referenced Julia function with the given parameters as general objects.
- `JuliaValue Invoke()`
    - Calls a referenced Julia function that doesn't take any parameters.
- `(JuliaFunction) JuliaValue`
    - An explicit operator that converts a JuliaValue into a JuliaFunction.
- `JuliaFunction GetJuliaFunction(string name)`
    - Gets the reference of a Julia function and safes the reference inside a JuliaFunction.
- `JuliaFunction GetJuliaFunctionsByModule(string name, string moduleName)`

    - Gets the reference of Julia function that is part of a module and safes the reference inside a JuliaFunction.

JuliaModule

- `JuliaModule(IntPtr wrapped)`
    - A Constructor that takes an IntPtr.
- `JuliaModule(JuliaValue juliaValue)`
    - A Constructor that takes a JuliaValue and converts it into a JuliaModule.
- `(JuliaModule) JuliaValue`
    - An explicit operator that converts a JuliaValue into a JuliaModule.
- `GetJuliaModule(string moduleName)`
    - Gets the reference of a Julia module and safes the reference inside a JuliaModule.

Type

Type is for checking and handling different data types.

Array

Array can be used to represent Julia arrays.

When there is no need for manipulating the array's data inside of Julia, it is recommended to translate the Julia arrays into C# arrays using the `AsArrayND()` method of the JuliaValue struct.

JuliaInstallationManager:

The class responsible for testing the Julia installation and providing an easy-to-use solution for setting up the machine for using Julia.

- `void SetupInstallation()`
    - This method is responsible for setting up the machine for using Julia. First a Julia test is performed to test if the communication with Julia is functioning. If the test fails an installation of Julia version 1.6.3 is searched on the machine. If necessary, required environment variables will be set and defined, or you will be asked to download and install Julia version 1.6.3.
- `void DefineJuliaInstallationPath()`
    - This method changes the set path of the Julia installation directory if a valid path is passed. It can be used to set the Julia installation path to a directory that isn't located on Julia's standard installation path.
- `void DownloadJulia()`
    - This will download the installer exe for Julia version 1.6.3.
- `void CheckJuliaInstallation`
    - This method searches for the Julia installation on the machine and on the PATH, and it also searches the newly defined environment variable JULIA_DIR and returns the found path and values as one single string.

JuliaModelHandler:

The JuliaModelHandler offers the user an inspector-oriented MonoBehaviour with two serialized arrays for parsing the names of all needed Julia modules and functions that will be referenced on start-up. The

references are then stored in a dictionary and can then be called with their respective names as keys.

- `void LoadModelDependencies()`
  - Includes the Julia files listed in the Inclusions.txt file into the current Julia process. This allows using the contained functions and modules of those files.
- `JuliaValue InvokeReferencedFunction(string functionName, JuliaBase.JuliaValue{} parameters)`
  - Calls a referenced Julia function that is saved in the ModelHandler's dictionary with the given parameters as JuliaValues.
- `JuliaValue InvokeReferencedFunction(string functionName, object{} parameters)`
  - Calls a referenced Julia function that is saved in the ModelHandler's dictionary with the given parameters as general objects.
- `JuliaValue InvokeReferencedFunction(string functionName)`
  - Calls a referenced Julia function that is saved in the ModelHandler's dictionary and that doesn't take any parameters.

FileHandler:

- `GetInclusions()`
  - Returns the names of the Julia files referenced in the Inclusions.txt file as a String array.

IjuliaPluginDebugger:

This Interface needs to be implemented by a MonoBehaviour. This MonoBehaviour then needs to be passed to the JuliaInstallationManager in the inspector as its "Julia Debugger Implementation", allowing the InstallationManager to display messages outside the console.

- `DisplayMessage();`
  - This method definition is meant for logging and debugging normal messages.
- `DisplayWarning();`
  - This method definition is meant for logging and debugging warnings.
- `DisplayError();`
  - This method definition is meant for logging and debugging errors.

JuliaBaseManager:

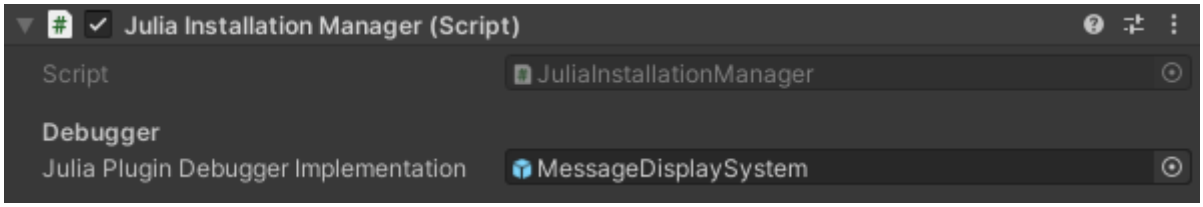
This MonoBehaviour is used for loading and referencing the necessary helper functions for handling multi-dimensional Julia arrays at the start-up of a scene.

MonoBehaviours in the Scene

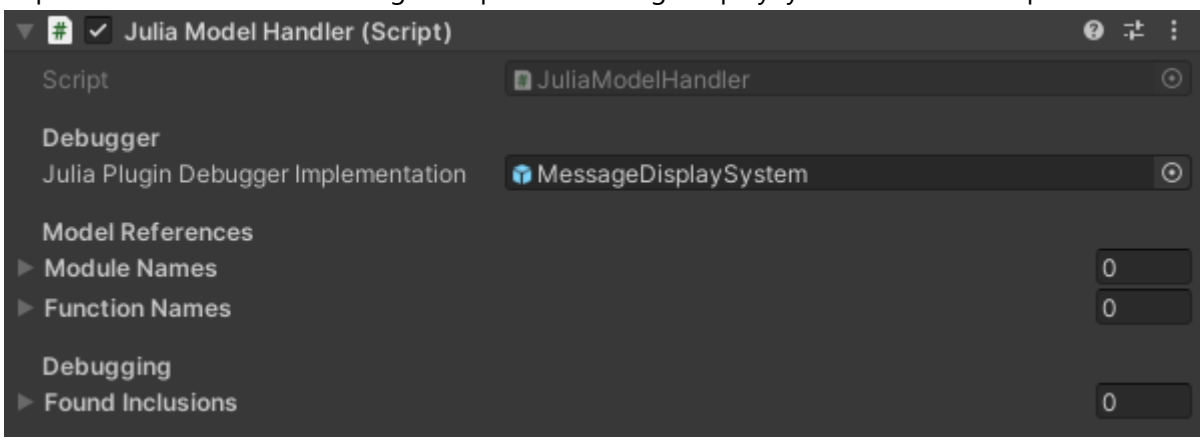
In order to use Julia during runtime some of the above-mentioned MonoBehavoirs need to be placed inside the scene.

1. The JuliaInstallationManager should be placed in your starting scene to test and set up the Julia installation if necessary. This instance should also receive an instance of your IJuliaPluginDebugger

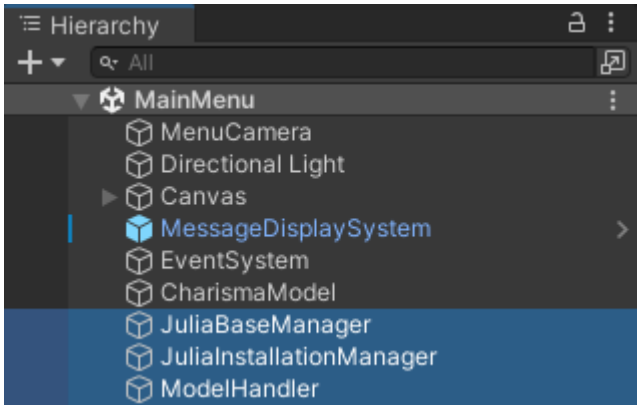
implementation. In the following example the MessageDisplaySystem is such an implementation:



2. The JuliaBaseManager should be placed in your starting scene so that the necessary helper functions for handling multi-dimensional Julia arrays can be loaded and referenced.
3. It is recommended to implement the IJuliaPluginDebugger interface inside a class deriving from Monobehaviour. By doing this you can parse an instance of this class to the JuliaInstallationManager or the JuliaModelHandler inside the Inspector for logging and debugging occurring errors and warnings.
4. If you want to use an inspector-oriented solution for referencing Julia functions and modules you could place an instance of the JuliaModelHandler inside the scene. This is purely optional and based on your preferences. This instance should also receive an instance of your IJuliaPluginDebugger implementation. In the following example the MessageDisplaySystem is such an implementation:



- The following example is taken from the MGM Simulation and shows instances of JuliaBaseManager, JuliaInstallationManager, and JuliaModelHandler (here ModelHandler) in the scene hierarchy. The MessageDisplaySystem represents an example implementation of the IJuliaPluginDebugger interface:



Examples

The following section lists some C# examples of how the above-mentioned functions and structures can be used.

Functions

Getting Julia functions:

```
JuliaBase.JuliaFunction func;
func = new
JuliaBase.JuliaFunction(JuliaBase.Instance.EvalString("function_name"));
```

OR:

```
JuliaBase.JuliaFunction func = GetJuliaFunction("function_name");
```

Using Julia functions:

```
var value = func.Invoke();
```

OR with parameters:

```
var value = func.Invoke(new JuliaValue[] { });
```

Arrays

Working with arrays: Let's say `func.Invoke()` returns a JuliaValue pointing to a one-dimensional array in Julia. Type needs to be the C# type equivalent of the respective Julia type (f.e. Double and Float64).

```
var arrayReference = func.Invoke();
var array = juliaArray.AsArray1D<T>();
```

Working with multi-dimensional arrays: Let's say `func.Invoke()` returns a JuliaValue pointing to a two-dimensional array in Julia. Now we want to access the array in the third cell of dimension one and additionally, we want to save the value of type Float64 in the first cell of this array as a Double.

```
var multiDimArray = func.Invoke();
var  subArray = JuliaBase.Instance.GetFromDimOne(multiDimArray, 3);
double value = JuliaBase.Instance.GetFromDimTwo(multiDimArray, 3, 1).UnboxFloat64;
```

Working with matrices: Matrices in Julia are basically a special type of multi-dimensional arrays. When converting the data from Julia into C# the matrix needs to be translated into a one-dimensional C# array. Let's say `func.Invoke()` returns a JuliaValue pointing to a matrix in Julia filled with values of type Float64.

```
var matrix = func.Invoke();
var matrixAsArray = JuliaBase.Instance.MatrixToArray(matrix).AsArray1D<double>();
```

EvalString & EvalStringArray

EvalString: The method `EvalString(string message)` can be seen as a Julia console inside C#. As long as the passed string equals valid Julia code it will be evaluated as such. The result of this evaluation is then returned. This was for example used in the "Getting Julia functions" example. There the fact was abused, that typing the name of a loaded function inside Julia returns a reference of this function. But `EvalString(string message)` could also be used like this:

```
// sqrt() is the Julia function for calculating the square root
var sqrtNine = JuliaBase.Instance.EvalString("sqrt(9.0)").UnboxFloat64;
```

EvalStringArray: `EvalStringArray(string{ } array)` does basically the same as `EvalString(string message)` **without** returning anything. Its purpose is loading in multiple instructions on a single call. This is for example used for loading in the required Julia files from Inclusions.txt.

IJuliaPluginDebugger

Interface code:

```
public interface IJuliaPluginDebugger
{
    public abstract void DisplayMessage(string message);
    public abstract void DisplayWarning(string message);
    public abstract void DisplayError(string message);
}
```

Implementation example: Let's assume the MyDisplay class creates a pop-up message in your scene based on which Log methods is called.

```
using JuliaPlugin;

public class MyImplementation : MonoBehaviour, IJuliaPluginDebugger
{
    public void DisplayMessage(string message) {
        MyDisplay.Log(message);
    }
    public void DisplayWarning(string message) {
        MyDisplay.LogWarning(message);
    }
    public void DisplayError(string message) {
        MyDisplay.LogError(message);
    }
}
```

Loading dependencies

In order to use the code of the Julia files that are referenced in the Inclusions.txt file the LoadModelDependencies() method needs to be called. This could for example be done in one of your classes the Start() method.

```
using JuliaPlugin;

public class MyLoadingClass : MonoBehaviour
{
    void Start() {
        JuliaModelHandler.LoadModelDependencies();
    }
}
```