ModTool 1.3.0 Documentation

Overview

ModTool makes it easy to add mod support your game. It enables modders to use Unity to create scenes, prefabs and code and export them as mods for your game.

To make sure the modder's scripts or Assemblies will be compatible with the game, ModTool has a simple but powerful code validator, that lets you configure any number of restrictions and requirements.

ModTool creates a custom unitypackage for every game, which includes everything to create mods.

Features

- Let modders use the Unity editor to create scenes, prefabs and code for your game
- Scripts and assemblies are fully supported
- Code validation
- Supports Windows, OS X, Linux and Android.
- Mod conflict detection
- Automatic Mod discovery
- Asynchronous discovery and loading of mods.

Limitations

- ModTool relies heavily on AssetBundles, which means mods can only be created with the same version of Unity that was used for building the game. The exporter will check if the correct version is used and inform the user if that's not the case.
- Unity can't deserialize fields of serializable types that have been loaded at runtime. This means that a Mod can't use it's own serializable Types in the inspector. Serializable types that aren't loaded at runtime and are part of the game do work.
- Mods have to rely on the game's project settings. E.g. Mods can not define their own new tags, layers and input axes. The created Mod exporter will include the game's project settings.

Contact

If you have any questions, suggestions, feedback or comments, please do one of the following:

- Send me an email: tim@hellomeow.net
- Make a post in the ModTool thread on the Unity forums

Setting up ModTool

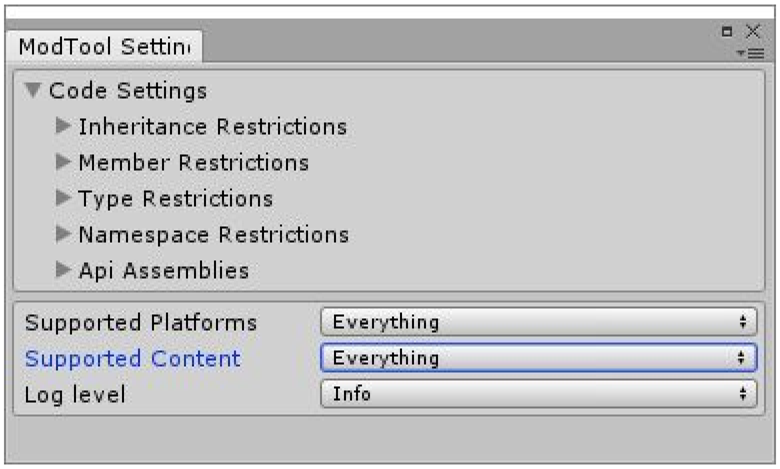
ModTool consists of two parts; the part that can export mods and the part that can find and load mods. The exporter is created automatically, on a per game basis. ModTool creates a Unity package containing everything that's needed to create mods for your game.

Importing ModTool

To begin using ModTool, just import the Unity Package. After that, you can start configuring it and generate a Mod exporter for your game. Use ModManager to find and eventually load mods. See the included example for a quick overview of the package.

Settings

In the settings window you can configure the things ModTool needs to know about the game.


ModTool Settin
▼ Code Settings
▶ Inheritance Restrictions
▶ Member Restrictions
▶ Type Restrictions
▶ Namespace Restrictions
▶ Api Assemblies

Supported Platforms	Everything
Supported Content	Everything
Log level	Info

Here you can set up restrictions (see Setting up Restrictions), the game's API assemblies you want to include, supported platforms and the types of content you allow to be included.

In API Assemblies you can include Assembly Names for the game's API assemblies. These should be DLL files that can provide some kind of interface to the game, or editor tools you want to include. The API Assemblies will be included in the created exporter and are also needed for code validation when loading the mod.

The Assembly names should be the Assembly's simple name. Not the display name, fully qualified name or file name. The simple name is the name that has been configured in the Visual Studio project. It's usually the same as the file name without the extension.

Setting up Restrictions

With Restrictions you can control the use of namespaces, Types, a Type's members, or inheritance.

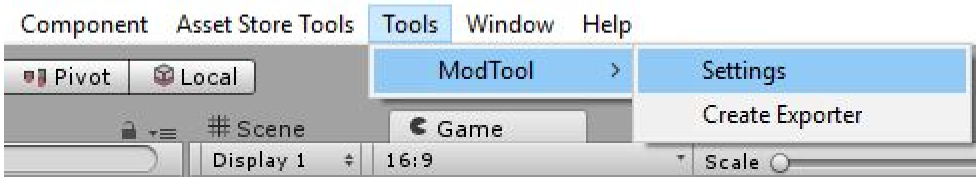
Restrictions can be applied to all Types, or Types that derive from a specific Type. Restrictions can either require or prohibit something.

Code will be validated before exporting a mod and before a mod is loaded into the game, to make sure mods will be compatible with the game and don't cause issues.

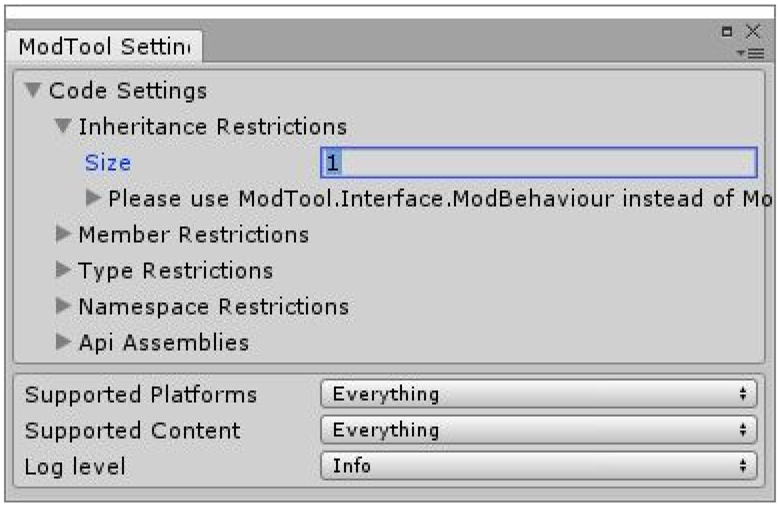
ModTool has a number of default restrictions that are included. These restrictions enforce the use of ModBehaviour and restrict some namespaces like System.IO and System.Reflection. These restrictions serve both as an example and to support ModTool.Interface They can be removed if they are not needed.

Adding A Restriction

To add a restriction, open ModTool's setting window by going to the ModTool menu.



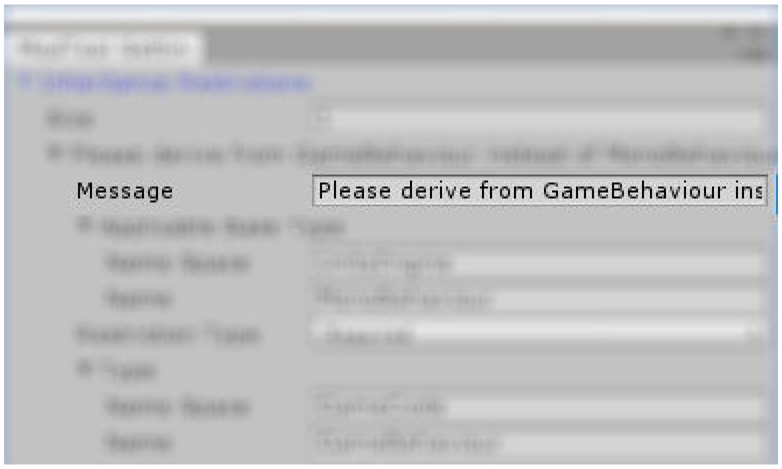
Here you can see various settings, including restrictions. Adding a Restriction works similarly to how you'd add a new axis in Unity's input manager. You can add a new Restriction by increasing the size of the collection.



ModTool Settin
Code Settings
Inheritance Restrictions
Size 1
Please use ModTool.Interface.ModBehaviour instead of Mo
Member Restrictions
Type Restrictions
Namespace Restrictions
Api Assemblies
Supported Platforms Everything
Supported Content Everything
Log level Info

Message

A Restriction's message is what will be displayed or logged when the loading or exporting of a Mod fails due to this Restriction.



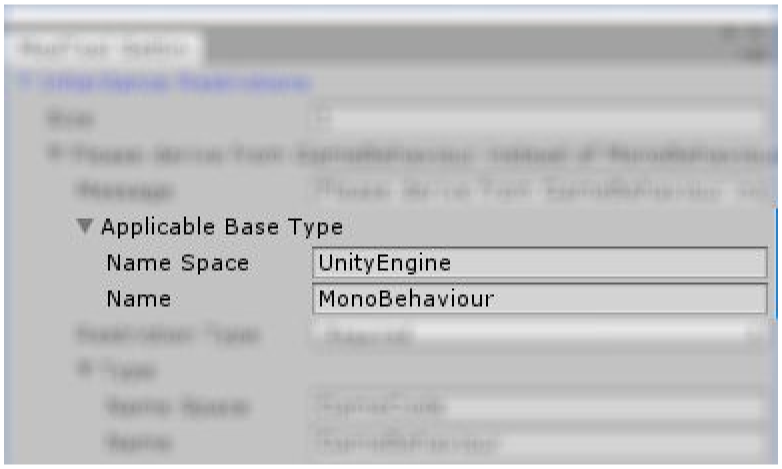
Message: Please derive from GameBehaviour ins

Applicable Base Type

The applicable base type of a restriction is the base type to which the Restriction will be applied. For example, if you want a restriction to only affect Types that derive from MonoBehaviour, you can configure it here.

When left empty, the Restriction will be applied to all Types.

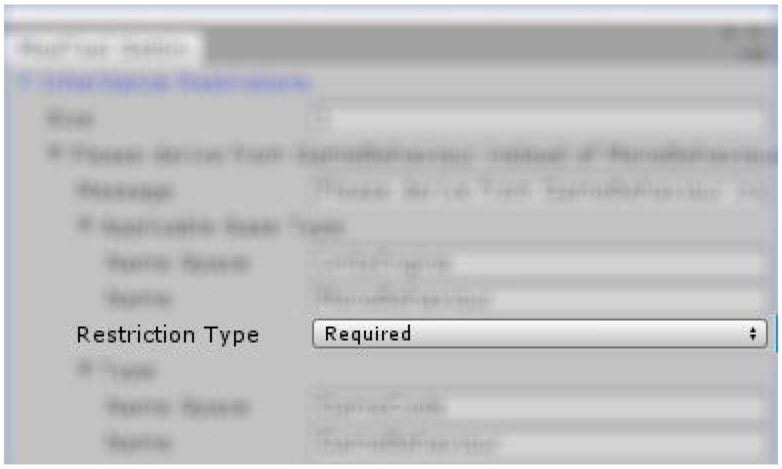
This restriction will be applied to all Types that derive from MonoBehaviour.



Name Space: UnityEngine
Name: MonoBehaviour

Restriction Mode

Use the Restriction's RestrictionMode to determine whether the Restriction prohibits or requires the use of something.



Restriction Type: Required

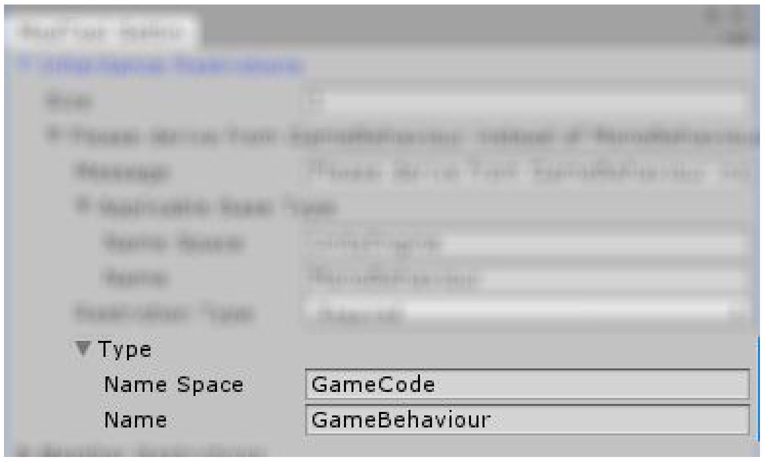
Restriction

For each Restriction you need to provide which member, Type, namespace or base class you want to prohibit or require.

Note:

When you want to restrict the use of a property, you need to provide the name of the generated getter and/or setter methods. Usually these are "get_" or "set_" followed by the property name. Constructors can be restricted by using ".ctor".

This restriction will require that all Types that derive from MonoBehaviour also derive from GameCode.GameBehaviour. This is a way to make sure the modder will use GameBehaviour instead of MonoBehaviour.



Type
Name Space: GameCode
Name: GameBehaviour

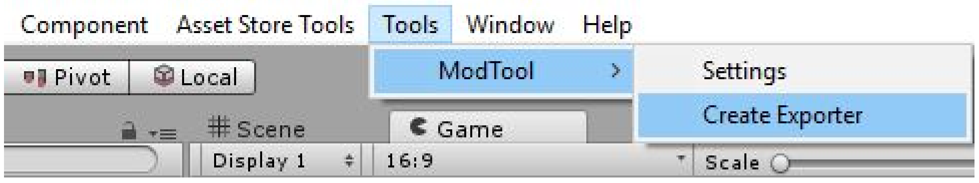
Creating an exporter

The Mod Exporter is the package modders will use to export their assets and code, so they can be used as Mods in your game.

The package includes everything that's needed to create mods for your game. Mainly the ModTool exporter dll that has been generated for your game, some parts of ModTool, and the game's api dll's that have been added in the ModTool settings window.

Creating the Mod Exporter package

The package will be automatically generated and included in the game's directory when you build the game in Unity. It can also be created at any time by using the ModTool menu.



The package will be named **"<productName> Mod Tools.unitypackage"**.

Test Game Mod Tools.unitypackage Unity package file 22 KB

ModTool relies heavily on AssetBundles, which means mods can only be created with the same version of Unity that was used for building the game. The exporter will check if the correct version is used and inform the user if that's not the case.

Unity can only use Components in interdependent assemblies when all of the assemblies are placed in the same folder. Because of this, ModTool will move all API assemblies to the ModTool folder before creating the exporter.

Finding Mods

ModTool will look for mods automatically. It does this by monitoring one or more directories for any added, changed or removed mods.

ModTool's default search directory is the "Mods" folder inside the game's install directory. On Android, this is based on Application.persistentDataPath. It's possible to add or remove search directories.

ModTool keeps a collection of available Mods and provides events for whenever a Mod is added, changed or removed.

Search Directories

To add a search directory, use ModManager.AddSearchDirectory(String)

To remove a search directory, use ModManager.RemoveSearchDirectory(String)

The search directories can be refreshed with ModManager.RefreshSearchDirectories(). This will look for any added, removed or changed mods. ModManager.refreshInterval configures an automatic refresh interval.

When a new Mod is found, or when a Mod has been removed, ModTool will update the collection of available Mods.

Mods

ModManager. mods contains all Mods that are currently available. The ModManager. ModsChanged event will be triggered when this collection changes.

ModManager. ModFound will occur when a new Mod has been found. ModManager. ModRemoved will occur when an existing mod has been removed.

Mod.modInfo contains all of a Mod's info, like the kinds of content that is included in the Mod, the Mod's name and the Mod's version.

A Mod instance will become invalid when a change is detected in its folder. For example, when a Mod's files are removed. Make sure that you're not using old Mod instances that are marked invalid. Invalid Mods won't load.

Loading a Mod

Available Mods can easily be loaded and unloaded. Loading a mod loads the Mod's Assemblies and assets so they can be used in the game.

Loading Mods can be done asynchronously. Both the ModManager and Mods provide events for when they're loaded, unloaded or when async loading has been cancelled.

Loading and Unloading

Load a mod with Mod.Load() or Mod.LoadAsync(). When a Mod has finished loading, Mod.Loaded and ModManager.ModLoaded will occur.

To unload a mod, use Mod.Unload(). Because Mods can be loaded asynchronously, a Mod could still be loading when Unload is called. In this case, unloading will be queued until it can be unloaded, or when Load is called again. When a Mod has unloaded, Mod.Unloaded and ModManager.ModUnloaded will occur.

Mod.isEnabled can be used to keep track of which mods a user wants to enable or disable. This property does not affect what you can do with a Mod; a Mod that is not enabled can still be loaded. This property is saved in the Mod's modInfo. With Mod.ConflictingModsEnabled() you can check if any conflicting mods are enabled as well. A Mod can not be loaded when another conflicting Mod is loaded.

Loaded Mods

Once a Mod has been loaded, you can use its resources. These are the prefabs, scenes and Types in any of the Mod's Assemblies.

Scenes are wrapped in the ModScene class, to make them easy to handle and load in a similar way as Mods.

Included prefabs are accessible through Mod.prefabs. Prefabs should be instantiated through the Mod's contentHandler. This will initialize any scripts on the prefab.

When a Mod is unloaded while its scenes are still loaded, the scenes will be unloaded. Instances of prefabs and components that are still in use will be destroyed.

It is a good practice to make sure instances of prefabs and GameObjects created by the Mod are accounted for by either the modder or the game. By default, ModTool will enfore the use of it's own Instantiate and AddComponent methods that make sure this gets handled.

Creating a Mod

Mods are created in Unity. You can create scenes, prefabs and scripts just like you would in any other Unity project.

Once you have finished creating the assets and scripts, you can export the project as a Mod. After that, they can be used in the game.

Mod Project

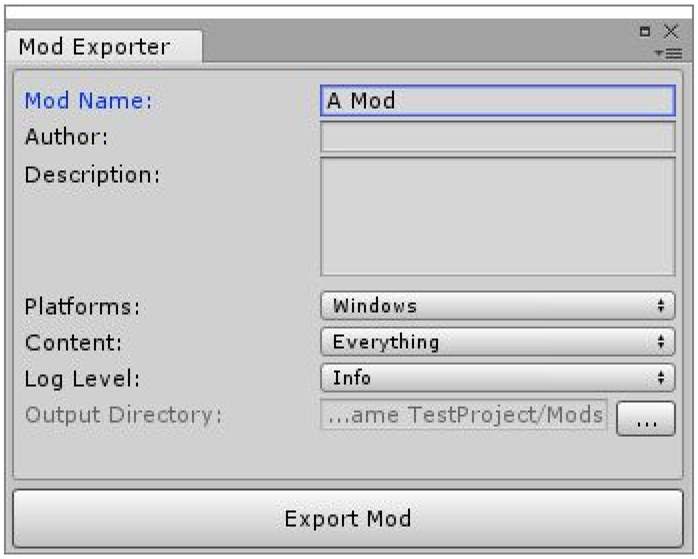
To begin creating a Mod, create a new Unity project. It is recommended to have a separate project for each Mod.

The Mod tools for the game are included as a .unitypackage file. You can import the package into Unity and start creating Mods.

Exporting a Mod

To export the project as a mod, go to the ModTool menu and select Export Mod.

Here you can give the Mod a name and other information. You can select which platforms the Mod will support and whether to include scenes and prefabs.



Mod Exporter

Mod Name:	A Mod
Author:	
Description:	
Platforms:	Windows
Content:	Everything
Log Level:	Info
Output Directory:	...ame TestProject/Mods

ModTool relies heavily on AssetBundles, which means mods can only be created with the same version of Unity that was used for building the game. The exporter will check if the correct version is used and inform you if that's not the case.

Restrictions

The game might have some restrictions imposed on what you can do with scripting. This is to make sure it will be compatible with the game and to prevent the Mod from breaking things as much as possible.

The mod will be validated before it's exported. You can also choose to validate it at any time to see if your mod is compatible with the game. Any errors during validation will be logged to the console or the game's log file.

ModBehaviour

By default, ModTool will force the use of ModTool.Interface.ModBehaviour instead of MonoBehaviour. ModBehaviour is a MonoBehaviour, but with some added functionality that enables ModTool to keep track of what a Mod instantiates.

When a Mod's scripts derive from ModBehaviour, any remaining GameObjects can be cleaned up when the Mod is unloaded.