3rd Party Plugins

In Unity, you normally use **scripts** to create functionality, but you can also include code created outside Unity in the form of a **plug-in**. You can use two different kinds of plug-in in Unity:

- **Managed plug-ins**: managed .NET assemblies you can create with tools like Visual Studio. They only contain .NET code, which means they can't access any features that the .NET libraries do not support. For more information, see Microsoft's documentation of managed code.
- **Native plug-ins**: platform-specific native code libraries. They can access features like operating system calls and third-party code libraries that would otherwise be unavailable to Unity.

Managed code is accessible to the standard .NET tools that Unity uses to compile scripts. The only difference between managed plug-in code and Unity script code is that the plug-ins are compiled outside of Unity and so Unity might not be able to access the source. When using native plug-ins, Unity's tools can't access third-party code libraries in the same way that they can access the managed libraries. For example, if you forget to add a managed plug-in file to the project, you will get standard compiler error messages. Whereas, if you forget to add a native plug-in file to the project, you will only see an error report when you try to run the project.

The following pages explain how to create and use plug-ins in your Unity Projects:

▼ Managed plug-ins

Managed **plug-ins** are .NET assemblies you create and compile outside of Unity, into a dynamically linked library (DLL) with tools such as Visual Studio.

This is a different process from standard C# **scripts**, which Unity stores as source files in the Assets folder in your Unity project. Unity compiles standard C# scripts whenever they change, whereas DLLs are pre-compiled and don't change. You can add a compiled .dll file to your project and attach the classes it contains to **GameObjects** in the same way as standard scripts.

For more information about managed code in C#, see Microsoft's What is managed code? documentation.

Managed plug-ins contain only .NET code, which means they can't access any features that the .NET libraries don't support. However, managed code is accessible to the standard .NET tools that Unity uses to compile scripts.

When you work with DLLs in Unity, you must complete more steps than when you work with scripts. However, there are situations where you might find it helpful to create and add a .dll file to your Unity project instead, for example:

- You want to use compilers in your code that Unity doesn't support.
- You want to add third-party .NET code in a .dll file.
- You want to supply code to Unity without the source.

This page explains a general method you can use to create **managed plug-ins**, as well as how you can create managed plug-ins and set up a debug session using Visual Studio.

Creating a managed plug-in

To create a managed plug-in, you need to create a DLL. To do this, you need a suitable compiler, such as:

- Visual Studio
- MsBuild
- .NET SDK

Not all compilers that produce .NET code are compatible with Unity, so you should test the compiler with some available code before doing significant work with it. The method you use to create a DLL depends on if the DLL contains Unity API code:

- If the DLL contains Unity API code, you need to make Unity's own DLLs available to the compiler before compiling:
    1. To find the Unity DLLs:
        - On Windows, go to: `C:\Program Files\Unity\Hub\Editor\<version-number>\Editor\Data\Managed\UnityEngine`
        - On macOS:

1. Find the `Unity.app` file on your computer. The path to the Unity DLLs on macOS is: `/Applications/Unity/Hub/Editor/<version-number>/Unity.app/Contents/Managed/UnityEngine`
2. Right click on `Unity.app`
3. Select **Show Package Contents**.

2. The `UnityEngine` folder contains the .dll files for a number of modules. Reference them to make them available to your script. Some namespaces also require a reference to a compiled library from a Unity project (for example, `UnityEngine.UI`). Locate this in the project folder's directory: `~\Library\ScriptAssemblies`

- If the DLL does not contain Unity API code, or if you've already made the Unity DLLs available, follow your compiler's documentation to compile a .dll file.The exact options you use to compile the DLL depend on the compiler you use. As an example, the command line for the Roslyn compiler, `csc`, might look like this on macOS:

```
csc /r:/Applications/Unity/Hub/Editor/<version-number>/Unity.app/Contents/Managed/UnityEngine.dll /target:library /out:MyManagedAssembly.dll /recurse:*.cs
```

In this example:

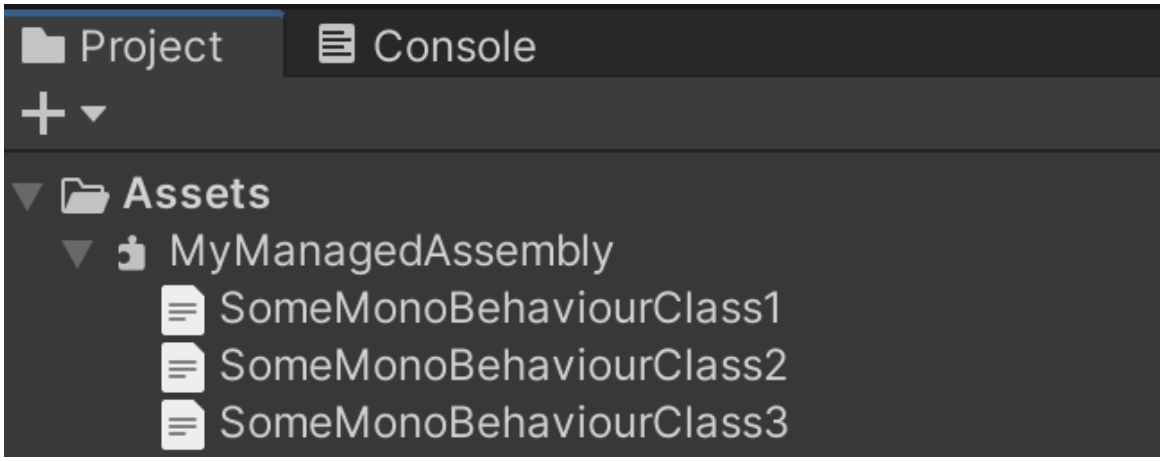
- Use the `/r` option to specify a path to a library to include in the build, in this case, the `UnityEngine` library.
- Use the `/target` option to specify the type of build you require; "library" signifies a DLL build.
- Use `/out` to specify the name of the library, which in this case is "MyManagedAssembly.dll".
- Add the name of the source files to be included. Use the `/recurse` method to add all the files ending in ".cs'' in your current working directory and any subfolders. The resulting .dll file appears in the same folder as the source files.

Using a managed plug-in

After you've compiled the DLL, you can drag the .dll file into the Unity project like any other asset. You can then:

- Expand the managed plug-in to view the separate classes inside the library.

- Drag classes that derive from MonoBehaviour onto Game Objects.
- Use non-MonoBehaviour classes directly from other scripts.



An expanded DLL with the classes visible

Create a DLL with Visual Studio

This section explains:

- How to build and integrate a simple DLL example with Visual Studio
- How to prepare a debugging session for the DLL in Unity.

Setting up the Project

1. Open Visual Studio and create a new project.
2. Select **File** > **New** > **Project** > **Visual C#** > **.Net Standard** > **Class Library (.NET Standard)**.
3. Add the following information to the new library:
    - **Name**: The namespace (for this example, use `DLLTest` as the name).
    - **Location**: The parent folder of the project.
    - **Solution name**: The folder of the project.
4. Make the Unity DLLs available to your script. In Visual Studio, open the contextual menu for **References** in the Solution Explorer and select **Add Reference** > **Browse** > **Select File**.
5. Select the required .dll file, located in the UnityEngine folder.

Coding the DLL

1. For this example, rename the default class to `MyUtilities` in the Solution browser.
2. Replace its code with the following:

```
using System;
using UnityEngine;

namespace DLLTest {

    public class MyUtilities {

        public int c;

        public void AddValues(int a, int b) {
            c = a + b;
        }

        public static int GenerateRandom(int min, int max) {
            System.Random rand = new System.Random();
            return rand.Next(min, max);
        }
    }
}
```

1. Build the project to generate the DLL file along with its debug symbols.

Debugging a DLL in Unity

To set up a debugging session for a DLL in Unity:

1. Create a new project in Unity and copy your built .dll file (for example, `<project folder>/bin/Debug/DLLTest.dll`) into the Assets folder.
2. Create a C# script called Test in the Assets folder.
3. Replace its contents with a script that creates a new instance of the class you created in your DLL, uses its functions and displays the output in the **Console** window. For example, to create a test script for the DLL from the section above, copy the following code:

```
using UnityEngine;
using System.Collections;
using DLLTest;

public class Test : MonoBehaviour {
```

```
      void Start () {
         MyUtilities utils = new MyUtilities();
         utils.AddValues(2, 3);
         print("2 + 3 = " + utils.c);
      }

      void Update () {
         print(MyUtilities.GenerateRandom(0, 100));
      }
}
```

1. Attach this script to a GameObject in the **Scene** and click Play.

Unity displays the output of the code from the DLL in the Console window

▼ Native plug-ins

Unity supports native **plug-ins**, which are libraries of native code you can write in languages such as C, C++, and Objective-C. Plug-ins allow the code you write in C# to call functions from these libraries. This feature allows Unity to integrate with middleware libraries or existing C/C++ code.

The **native plug-in** provides a simple C interface, which the C# script then exposes to your other **scripts**. Unity can also call functions that the native plug-in exports when certain low-level rendering events happen (for example, when you create a graphics device). See Low-level native plug-in interface for more information.

For an example of a native plugin, see Native Renderer Plugin.

Using a native plug-in

To use a native plug-in:

1. Write functions in a C-based language to access the features you need.
2. Compile them into a library.
3. In Unity, create a C# script that calls functions in the native library.

You build native plug-ins with native code compilers on the target platform. Because plug-in functions use a C-based call interface, you must declare the functions with C linkage to avoid name mangling issues.

Example

A simple native library with a single function might have code that looks like this:

`float ExamplePluginFunction () { return 5.0F; }`

To access this code from within Unity, use the following C# script:

```
using UnityEngine;
using System.Runtime.InteropServices;

class ExampleScript : MonoBehaviour {
    #if UNITY_IPHONE// On iOS plugins are statically linked into
    // the executable, so we have to use __Internal as the
    // library name.
    [DllImport ("__Internal")]
    #else// Other platforms load plugins dynamically, so pass the
    // name of the plugin's dynamic library.
    [DllImport ("PluginName")]
    #endif
    privatestaticexternfloat ExamplePluginFunction ();

void Awake ()
{
    // Calls the ExamplePluginFunction inside the plugin
    // And prints 5 to the console
    print (ExamplePluginFunction ());
   }
}
```