Quick Start

> ! This tutorial assumes that you have a fair grasp of the C# programming language and basic knowledge of Visual Studio and Unity Engine.

Visual Studio Template

An automated Visual Studio 2022 template for creating MelonLoader mods and plugins is available.
It handles the creation of the required boilerplate (the `MelonMod` / `MelonPlugin` class, `MelonInfo` , and `MelonGame`) as well as referencing the required assemblies for mod development, mainly MelonLoader, Harmony, and for Il2Cpp, proxy assemblies and the unhollower (Il2CppAssemblyUnhollower or Il2CppInterop). It also handles variation between MelonLoader or Unity versions, such as framework versions or override changes.

1. Download MelonLoader to your game and run it once before continuing.
2. Download the VSIX from the **GitHub repo**.
3. Close all instances of Visual Studio and run the VSIX installer (double-clicking it should open it).
4. Open Visual Studio and create a new project.
5. Search for `MelonLoader` and click on either Mod or Plugin.

6. Enter the project info and press Create.
7. Select the EXE of the game you are modding and press Open.
8. Wait for the project creation and it should open a Visual Studio window with a working project.

You may want to change the author in the `MelonInfo` attribute. It defaults to your computer's username.

Basic mod setup

First, you will need to create a new project. Unity versions require different project templates:

- Any `Il2Cpp` game:
  - Template: `Class Library`
  - Framework: `.NET 6.0`
- `Mono` game after or on `Unity 2021.2`:
  - Template: `Class Library`
  - Framework: `.NET Standard 2.1`
- `Mono` game after or on `Unity 2018.1`:
  - Template: `Class Library (.NET Framework)`
  - Framework: `.NET Framework 4.7.2`
- `Mono` game after or on `Unity 2017.1`:
  - Template: `Class Library (.NET Framework)`
  - Framework: `.NET Framework 3.5` or `.NET Framework 4.7.2` (depending on the game)
- Any other version:

- Template: `Class Library (.NET Framework)`
- Framework: `.NET Framework 3.5`

> Please note that `Class Library` and `Class Library (.NET Framework)` are different templates.

Doing so will create a new empty cs file, called `Class1`. This will be our mod main class.
I'll rename it `MyMod`. You can change it to whatever you would like though.

You will now need to reference the main MelonLoader assembly. Right click the `Reference` directory, `Add a reference...`, and click `Browse`.
Find to the folder of the game you installed MelonLoader on. The file you need to reference from here is `MelonLoader/MelonLoader.dll`.

Any type of mod requires a `MelonInfo` assembly attribute for MelonLoader to know what mod it's loading.
Additionally, a `MelonGame` assembly attribute is needed so MelonLoader knows what games your mod supports.
To set one up, go to the `Properties` directory and add these 3 lines to `AssemblyInfo.cs` or `Program.cs`:

cs

```cs
using MelonLoader;
using MyProject; // The namespace of your mod class
// ...
[assembly: MelonInfo(typeof(MyMod), "My Mod Name", "version", "Auth
[assembly: MelonGame("Game Developer", "Game Name")]
```

`MelonInfo` contains 4 required parameters and an optional one:

- `MyMod` : The main class of your mod. We will talk about it later
- `My Mod Name` : The name of your mod
- `version` : The version of the mod. It should respect the **semver format** (example: `1.0.0`)
- `Author Name` : The name of author of the mod
- `Download Link` : The link to download or find the mod [optional]

`MelonGame` contains 2 optional parameters:

- `Game Developer` : The developer of the game
- `Game Name` : The name of the game

Both parameters in `MelonGame` can be left empty to have MelonLoader not check them.
Omitting the `Game Name` parameter makes the mod load on all games made by the same developer, and omitting both parameters makes the mod load on all games.

Often the name of the game and developer doesn't match the marketing. To get guaranteed correct information find the `app.info` file in the `<Game Name>\<Game Name>_Data\` folder.

The MelonMod class

We are almost ready. Let's go back to our `MyMod` class and turn it into a Melon. First, let's add `using MelonLoader;` and make our `MyMod` class inherit from `MelonMod`.

At this point, your `MyMod` class should looks like this:

cs

```cs
using MelonLoader;

namespace MyProject
{
    public class MyMod : MelonMod
    {

    }
}
```

Your mod is now a valid Melon and can be loaded by MelonLoader.
In the following paragraphs, you will learn how to add some functionality to it.

Melon Callbacks

MelonMod comes with a few virtual callbacks that can be overridden:

- `OnEarlyInitializeMelon` : Called when the Melon is registered. Executes before the Melon's info is printed to the console.
  This callback may run before the Support Module is loaded.
  Do not reference any game/Unity members in this callback or override `OnInitializeMelon` instead.
- `OnInitializeMelon` : Called after the Melon was registered. This callback waits until MelonLoader has fully initialized
  It is safe to make any game/Unity references from and after this callback.
- `OnLateInitializeMelon` : Called after `OnInitializeMelon` . This callback waits until Unity has invoked the first 'Start' messages.
- `OnDeinitializeMelon` : Called before the Melon is unregistered. Also called before the game is closed.
- `OnUpdate` : Called once per frame.
- `OnFixedUpdate` : Called every 0.02 seconds, unless `Time.fixedDeltaTime` has a different value. It is recommended to do all important Physics loops inside this Callback.
- `OnLateUpdate` : Called once per frame after all `OnUpdate` callbacks have finished.
- `OnGUI` : Called at every **IMGUI** event. Only use this for drawing IMGUI Elements.
- `OnApplicationQuit` : Called when the game is told to close.
- `OnSceneWasLoaded` : Called when a new Scene is loaded.
- `OnSceneWasInitialized` : Called once the active Scene is fully initialized.

- `OnSceneWasUnloaded` : Called once a Scene unloads.
- `OnPreferencesSaved` : Called when Melon Preferences get saved.
- `OnPreferencesLoaded` : Called when Melon Preferences get loaded.

The following example shows how to implement those callbacks:

cs

```cs
using MelonLoader;

namespace MyProject
{
    public class MyMod : MelonMod
    {
        public override void OnSceneWasLoaded(int buildIndex, strir
        {
            LoggerInstance.Msg($"Scene {sceneName} with build indez
        }
    }
}
```

Melon Events

Some callbacks mentioned in the previous paragraph are just shorthand for MelonLoader's global MelonEvents.

MelonEvents are special events that Melons can subscribe to without having to worry about deinitialization.

MelonEvents were programmed to automatically dispose any subscribtions from deinitialized Melons.

The advantage of using MelonEvents instead of virtual callbacks is the ability to subscribe to events with a custom priority.
This is useful in cases your callback has to run earlier/later than a callback from any other mod.
Another advantage of MelonEvents is the ability to subscribe to events in other classes which can help keeping a cleaner mod structure.

Most global MelonEvents can be found in the public `MelonLoader.MelonEvents` class.

> The following example references the `UnityEngine.IMGUIModule` assembly.

This example shows how to draw a GUI element on top of most other mods through a MelonEvent:

cs

```cs
public class MyMod : MelonMod
{
    public override void OnInitializeMelon()
    {
        MelonEvents.OnGUI.Subscribe(DrawMenu, 100); // The higher t
    }

    private void DrawMenu()
```

```
        {
            GUI.Box(new Rect(0, 0, 300, 500), "My Menu");
        }
    }
```

Logging

In modding, logging is very important for making the mod users aware of what your mod is doing, but also for diagnosing issues.
Fortunately, MelonLoader has it's own logging system which is also available to mods.

Any `MelonMod` has it's own logger instance which can be accessed through the `LoggerInstance` property:

cs

```
public override void OnInitializeMelon()
{
    LoggerInstance.Msg("Hello World!");
}
```

In order to access the logger instance from a static method or from another class, you'll have to use a singleton for your mod class.
Fortunately, MelonLoader comes with a generic singleton class for mod classes, namely `Melon<T>` where `T` is your mod class.
Through `Melon<T>`, you can access your mod instance and your logger instance

from anywhere.

cs

```cs
public class MyMod : MelonMod
{
    public override void OnInitializeMelon()
    {
        HelloWorld();
    }

    public static void HelloWorld()
    {
        Melon<MyMod>.Logger.Msg("Hello World from a static method!'
    }
}
```

Assembly References

As seen in the previous `OnGUI` example, calling game/Unity methods is as simple as in a normal unity script.
However, compared to Unity, you have to reference all the game and Unity assemblies manually.

For games using the **Mono runtime**, all the game/Unity assemblies can be found in `[Game Directory]\[Game Name]_data\Managed\` .
For games using the IL2CPP runtime, all the game/Unity assemblies can be found in `[Game Directory]\MelonLoader\Managed\` (make sure you have ran the

game with MelonLoader at least once!).

Since IL2CPP converts all game assemblies to C++, MelonLoader is using **Il2CppInterop**, an IL2CPP proxy assembly generator which allows us to use IL2CPP assemblies from C#. Before we can use any assemblies generated by the Unhollower, it's required to reference the following assemblies first:

- `Il2Cppmscorlib.dll`
- `Il2CppInterop.Common.dll`
- `Il2CppInterop.Runtime.dll`

At this point, you're ready to make your first functional Melon.

Basic Mod Example

> The following example mod references the `UnityEngine.CoreModule`, `UnityEngine.InputLegacyModule` and `UnityEngine.IMGUIModule` assemblies.

This example mod allows the user to freeze and unfreeze the game by pressing the spacebar:

cs

```cs
using UnityEngine;
using MelonLoader;

[assembly: MelonInfo(typeof(TimeFreezer.TimeFreezerMod), "Time Free
```

```
namespace TimeFreezer
{
    public class TimeFreezerMod : MelonMod
    {
        private static KeyCode freezeToggleKey;

        private static bool frozen;
        private static float baseTimeScale;

        public override void OnEarlyInitializeMelon()
        {
            freezeToggleKey = KeyCode.Space;
        }

        public override void OnLateUpdate()
        {
            if (Input.GetKeyDown(freezeToggleKey))
            {
                ToggleFreeze();
            }
        }

        public static void DrawFrozenText()
        {
            GUI.Label(new Rect(20, 20, 1000, 200), "<b><color=cyan>
        }

        private static void ToggleFreeze()
        {
```

```
            frozen = !frozen;

            if (frozen)
            {
                Melon<TimeFreezerMod>.Logger.Msg("Freezing");

                MelonEvents.OnGUI.Subscribe(DrawFrozenText, 100); /
                baseTimeScale = Time.timeScale; // Save the origina
                Time.timeScale = 0;
            }
            else
            {
                Melon<TimeFreezerMod>.Logger.Msg("Unfreezing");

                MelonEvents.OnGUI.Unsubscribe(DrawFrozenText); // U
                Time.timeScale = baseTimeScale; // Reset the time s
            }
        }

        public override void OnDeinitializeMelon()
        {
            if (frozen)
            {
                ToggleFreeze(); // Unfreeze the game in case the me
            }
        }
    }
}
```