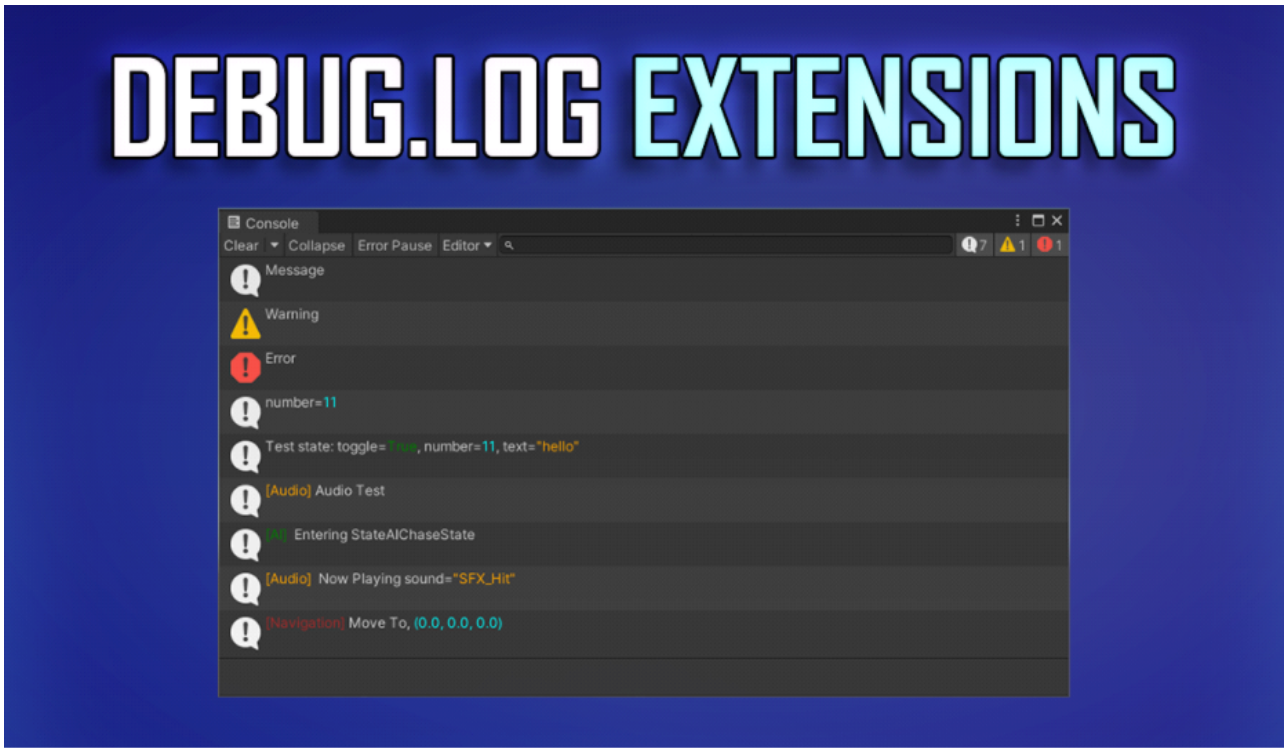

Documentation

Links

- **Asset Store**
- **Script Reference**
- **Unity Forum**

What Is a Debug.Log Extensions?

Debug.Log Extensions enhances the Debug class with numerous improvements that can greatly improve readability in the Console and save a lot of time by enabling more compact debugging code to be used.

This is a plug & play solution; all Debug commands in your classes will automatically switch to using the new and improved versions.

Installation

When you first install Debug.Log Extensions to a project you will be asked whether or not you would like to have it replace the default Debug class across the project.

If you choose "**Yes**" and some classes in your project contain the statement "*using Debug = UnityEngine.Debug;*" it can prevent the project from compiling.
To fix this you need to remove the statements from your code - they should no longer be necessary with Debug.Log Extensions installed.

Alternatively you can opt not to replace the default Debug class. In this case you will need to add the line "*using Debug = Sisus.Debugging.Debug;*" in any classes where you want to enable Debug.Log Extensions' enhancements.

Quick Start Guide

You can open an enhanced version of Console window called Console+ using the menu item **Window > Debugging > Console+**. This window is identical to the built-in Console window with the exception that it also contains a convenient dropdown menu for customizing which channels are currently enabled.

To enable the in-game GUI in builds either enable the Development Build flag in build settings or add *-log-gui-enable* as a launch parameter for your application.
The in-game GUI allows you customize enabled channels as well as view field values when Debug.DisplayOnScreen is used.

Feature Highlights

Clean Console

By default an empty line will be automatically added at the end of all your logged messages. This has the effect of pushing the stack trace off the main list in the Console, resulting in a cleaner look.

You can customize this behaviour in **Preferences > Console** using the preference item **Formatting > List Display Style**.

Syntax Highlighting

By default all your logged messages will be enhanced with colorful syntax highlighting.

You can customize this behaviour in **Preferences > Console** under the group **Colors**.

Automatic Context

The context Object is automatically determined for your messages whenever possible to enable locating message sources by clicking log entries in the Console.

Debug.LogChanges(()=>field)

This command can be used to print a message everytime that the value of the given field or property changes.

Note: the message will not be printed at the exact moment that the value changes, but at the end of the current frame - similar to how Debug.Break works.

Channels

If you prefix a log message with a tag inside square brackets you can now tie it to a specific channel. Here is an example**:**

```
Debug.Log(”[Audio]Playing ”, ()=>audioClip);
```

You can define new channels in **Project Settings > Console** by adding them to the Channels list. Here you can also customize their color in the Console view as well as whether or not the channel is enabled by default for all users.

Channels that you have added in the project settings will also automatically get compiled inside the Channel class. As an alternative to specifying the channel to use as a prefix of your message, you can also pass it as the first argument when calling various methods in the Debug class. This is functionally the same thing.

```
Debug.Log(Channel.Audio, Channel.Sfx, ”Playing ” + audioClip.name);
```

You can also enable or disable specific channels for yourself only in **Preferences > Console** by adding them to the **Whitelisted Channels** or **Blacklisted Channels** lists. These whitelists and blacklists take precedence over channel configuration in the project setting.

In addition you can customize channels at runtime using the in-game GUI. By default you can toggle the in-game GUI on and off using the **Insert** key. The default shortcut for all users can be customized in **Project Settings > Console** using the item **Toggle GUI** under **Shortcuts**. You can also override the shortcut for yourself only in **Preferences > Console**.

To enable channels programmatically you can use **Debug.EnableChannel** and disable them using **Debug.DisableChannel**. These can be used both at runtime and in edit mode. Note that changes made using these commands aren’t serialized, but are reset every time that scripts are reloaded in the editor or you close your build application.

Console+

An enhanced version of Console window called Console+ is also included. This window is a full replacement for the built-in Console window and contains enhancements such as a channel dropdown menu and the ability to find all instances of classes referenced in stack traces.

To open the window use the menu item **Window > Debugging > Console+ (Experimental)**.

Personal Channel

Each user has a personal channel derived from their username. This channel is automatically enabled only in the computer of the user in question.

This means that if you log messages using your personal channel others users of the project won't see them by default.

You can use the UniqueChannel property in the Dev class to learn what your personal channel is. This is also usually the first channel listed in the Channels dropdown in the Console+ view.

If your unique channel was for example "JohnDoe" then here is how you could use it to log a message that only visible to yourself by default:

```
Debug.Log("[JohnDoe]Test");
```

Debug.Log(()=>field)

This new command can be used to log both the name and value of a field or a property to the console.

This can handle null values as well collections without issues.

If you use this to log a field or a property belonging to an Object, then the context for the logged message will automatically be set to said Object.

Debug.LogState(target)

This command prints the full state of a target to the console. By default this will list all public instance fields and properties. You can customize this by using the variant that contains parameters includePrivate and includeStatic.

Debug.DisplayOnScreen(()=>field)

This command can be used to display the current value of a field on-screen. The value will be displayed both in the Scene view as well as the Game view via the in-game GUI.

Note: this will only work in the editor, in development builds or in non-development builds when using the launch parameter *-log-gui-enable*.

Debug.LogIf(condition, message)

The new LogIf methods function just like Debug.Log but only log a message when the provided condition is true.

A benefit of inlining the condition checking like this is that when build stripping is enabled in project settings or when using *Dev.LogIf* the condition check itself will also be fully stripped from release builds, ensuring that no CPU time is wasted on your debugging code.

```
public void PlayAudio(string fileName, Vector3 position)

{
	Dev.LogIf(debugEnabled, Channel.Audio, "PlayAudio(" + fileName + ") @ " + position);

	var clip = Resources.Load<AudioClip>("Audio/" + fileName);
	AudioSource.PlayClipAtPoint(clip, position);
}
```

Debug.Ensure(message)

Ensure methods combine condition checks with error logging upon failure. They also make sure an error is only logged once per session to avoid flooding the log file.

```
public void TransitionToScene(string sceneName)

{
	if(Debug.Ensure(!string.IsNullOrEmpty(sceneName))
	{
		TransitionToScene(SceneManager.GetSceneByName(sceneName));
	}
}
```

Debug.Guard(message)

Guard methods are just like Ensure methods but their return value is reversed, making them more fitting for situations where you want to return early upon failure.

```
public void TransitionToScene(string sceneName)

{
	if(Debug.Guard(!string.IsNullOrEmpty(sceneName), "Invalid sceneName"))
	{
		return;
	}
	TransitionToScene(SceneManager.GetSceneByName(sceneName));
}
```

Debug.LogToFile(message)

This can be used to log messages into custom text files instead of the console and the default log file.

By default the file will be created in the same directory as your default log but using the name *LogToFile.log*. You can specify a different name or change the path completely using the second *path* parameter.

By default your log files will be cleared with each new session when you call Debug.LogToFile for the first time. You can customize this behaviour by changing the third *clearFile* parameter. You can also use ClearLogFile to manually clear an existing log file.

Dev class

In addition to the extensions in the Debug class a new class called Dev is also introduced.

The Dev class contains many of the same methods as the Debug class such as Log, LogWarning, LogError, LogState, LogChanges, DisplayOnScreen and StartStopwatch.

These work identically to the methods found in the Debug class but any calls to them are completely omitted in release builds. This includes any string concatenation or method calls you do in the parameters too.

Note: to reference the Dev class in your scripts you need to include the line ”using Dev = Sisus.Debugging.Dev;” in your script.

Critical class

A new class called Critical is also introduced.

The Critical class contains many of the same methods as the Debug class such as Log, LogWarning, LogError, and Assert.

These function very similarly to the methods found in the Debug class but with three key differences:

1. The messages use a larger font in the console window.
2. They always include full stack trace even if stack traces have been disabled for other messages in Player Settings.
3. The messages are always recorded in builds even if "Use Player Log" is disabled in Player Settings.

Note: to reference the Critical class in your scripts you need to include the line ”using Critical = Sisus.Debugging.Critical;” in your script.