Documentation

For most up-to-date information
refer to the **Online Documentation**

Table of Contents

Installation

Package Manager

Init(args) can be installed using the Package Manager window inside Unity. You can do so by following these steps:

1. Open the Package Manager window using the menu item **Window > Package Manager**.
2. In the dropdown at the top of the list view select **My Assets**.
3. Find **Init(args)** in the list view and select it.
4. Click the **Download** button and wait until the download has finished. If the Download button is greyed out you already have the latest version and can skip to the next step.
5. Click the **Import** button and wait until the loading bar disappears and the Import Unity Package dialog opens.
6. Click **Import** and wait until the loading bar disappears. You should not change what items are ticked on the window, unless you know what you’re doing, because that could mess up the installation.

Init(args) is now installed and ready to be used!

Keeping Up To Date

You might want to check from time to time if Init(args) has new updates, so you don’t miss out on new features.

The process for updating Init(args) is very similar to installing it, and done using the Package Manager window inside of Unity.

You can do so by following these steps:

1. Open the Package Manager window using the menu item **Window > Package Manager**.
2. In the dropdown at the top of the list view select **My Assets**.
3. Find **Init(args)** in the list view and select it.
4. If you see a greyed out button labeled **Download**, that means that your installation is up-to-date, and you are done here! If instead you see a button labeled Update, continue to the next step to update Init(args) to the latest version.
5. Click the **Update** button and wait until the download has finished.
6. Click the **Import** button and wait until the loading bar disappears and the Import Unity Package dialog opens.
7. Click **Import**, then wait until the loading bar disappers. You should not change what items are ticked on the window, unless you know what you’re doing, because that could mess up the installation.

Init(args) is now updated to the latest version.

Referencing Init(args) Assemblies

To reference code in Init(args) from your own code you need to create assembly definition assets in the roots of your script folders and add references to the assemblies containing the scripts you want to use.

Init(args) contains three assemblies:

1. **InitArgs** – The main assembly that contains most classes including MonoBehaviour<T…>.
2. **InitArgs.Services** – This contains only a handful of classes related to service registeration: ServiceAttribute, EditorServiceAttribute and ServiceInitializer<T…>.
3. **InitArgs.Editor** – Editor-only assembly containing classes related to unit testing as well as custom editors and property drawers for the Inspector.

Assembly Setup

Most of your client components will only need to reference types found in the *InitArgs* assembly, so add a reference to that in your assembly definition assets as needed.

Additionally, any classes that you want to use global services, need to reference the InitArgs.Services assembly so that they can use the ServiceAttribute.

The InitArgs.Editor assembly you should only reference from your own editor-only assemblies. This might be useful when writing edit mode unit tests, if you want to make use of the Testable class to invoke non-public Unity event functions or the EditorCoroutine class to run coroutines.

Sisus.Init Namespace

To reference code in Init(args) you also need to add the following using directive to your classes:

```
using Sisus.Init;
```

Null Extensions

In all MonoBehaviour<T…> derived class you can have access to the Null property, which can be used for convenient null-checking of interface type variables, with support for identifying destroyed Objects as well:

```
if(interfaceVariable != Null)
```

If you want to be able to do the same thing in other types of classes, you can also import the Null property from the NullExtensions class:

```
using static Sisus.NullExtensions;
```

Demo Scene

The project comes with a demo scene containing a simple game showcasing many of the features offered by Init(args). It acts as an example of a very flexible and easily unit-testable architecture that can scale well to drive large projects as well.

Installing Dependencies

Before installing the demo scene you must first install the **Unity UI** and **Test Framework** packages.

To install the packages perform the following steps:

1. Open the Package Manager using the main menu item **Window > Package Manager**.
2. Click the second dropdown menu on the top bar and select **All packages**.
3. Find the Test Framework package in the list and select it.
4. Click the Install button and wait until the package has finished installing.
5. Next find the Unity UI package in the list and select it.
6. Click the Install button and wait until this package has also finished installing.

If you do not care to see the unit tests for the demo project, you can also skip installing the Test Framework package, and then untick the folder *Assets/Sisus/Init(args)/Demo/Tests* during installation of the Demo package.

Installing The Demo

To install the demo project perform the following steps:

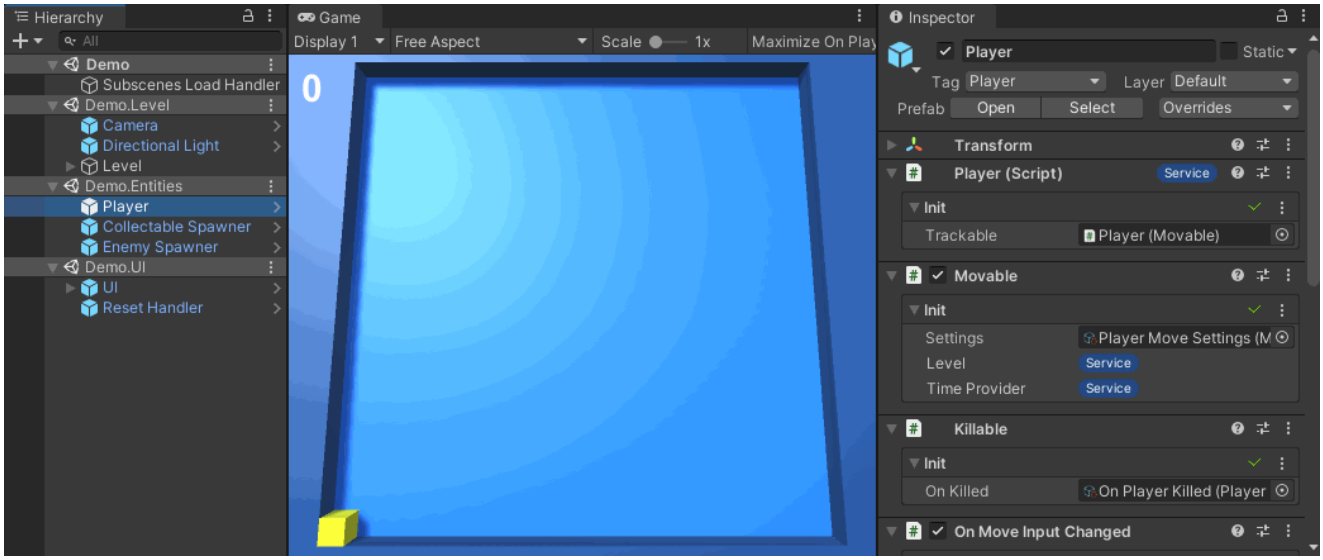
1.locate the unity package at *Assets/Sisus/Init(args)/Demo Installer* in the Project view and double-click it.

2.In the Import Unity Package dialog that opens select **Import**.

After installation has finished you can find the demo scene at *Assets/Sisus/Init(args)/Demo/Demo*.

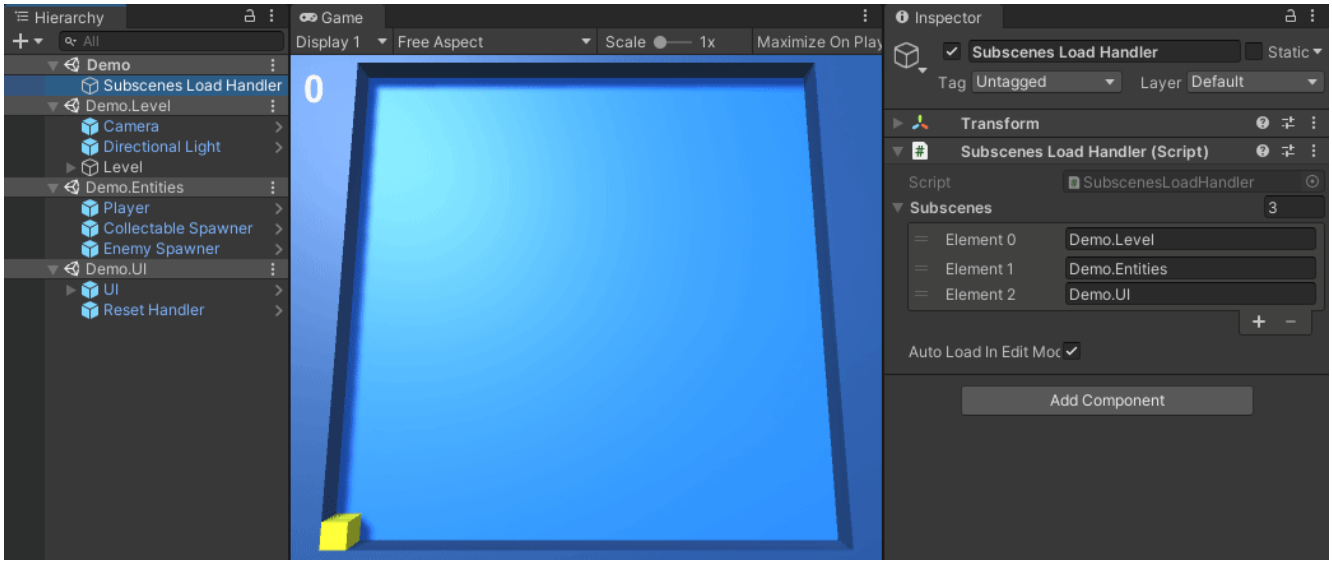
To try out the demo open the Demo Scene and enter Play Mode.

Gameplay



You control the yellow cube using the arrow keys and your objective is to collect as many green dots as you can without hitting any of the red cubes.

Scene Overview



The Demo has been split into four subscenes. This has been done to showcase how Init(args) can help enable a multi-scene workflow, which can help reduce the amount of conflicts that occur when multiple people working on the game at the same time.

1. **Demo**: The main scene just contains the Subscenes Load Handler which is responsible for loading all the other subscenes both in edit mode as well as play mode.
2. **Demo.Level**: Contains the camera, lighting as well as the GameObjects that make up the level.
3. **Demo.Entities**: Contains the Player object and two spawner objects responsible for creating collectables and enemies during gameplay.
4. **Demo.UI**: Contains the user interface for the game as well as the Reset Handler.

Some of the important objects in the demo include:

1. **Level**: The Level component defines the bounds of the level inside which the player can move and enemies and collectables can spawn. The class is also a service to make it more convenient for other objects to retrieve a reference to it. Nested underneath the GameObject are also the visual elements that make up the level.
2. **Player**: Contains the Player, Movable and Killable components. The Player component is responsible for detecting collisions with ICollectable object and the Killable component with IDeadly objects. The Movable component is responsible for moving the GameObject within the bounds of the level and is driven by the OnMoveInputChanged component.
3. **Enemy Spawner**: Contains the SpawnerInitializer which is used to specify the arguments for the SpawnerComponent. The SpawnerComponent is created at runtime, and is actually

just a simple wrapper for the Spawner, a plain old C# object, which actually holds all the logic for spawning the enemies.

4. **Collectable Spawner**: Just like Enemy Spawner except configured to spawn instances of the Collectable prefab instead of enemies.

5. **Reset Handler**: Houses the ResetHandler component which is used to reset the game state when the player presses the 'R' key.

6. **UI**: Contains the Score and Game Over texts. The Score texts has been hooked to update when the On Collected event is invoked and the Game Over text has been hooked to become active when the On Player Killed event is invoked. The events are represented by scriptablservicese object assets, so that objects in different scenes can reference it. The events are also, so that instances of the event triggers On Collected and On Player Killed can be used without needing to manually drag-and-drop a reference to the event asset. Both events also implement an interface which can be passed to clients to allow them to subscribe to the event and unsubscribe from it, but not trigger it, for better encapsulation.

All classes in the demo have XML documentation comments, and the best way to learn more about the demo project is to go examine its source code.

The demo also contains a number of unit tests to showcase how Init(args) can help make your code more easily unit testable. The tests contain some interesting patterns such as testing of coroutines in edit mode and suppression of logging for the duration of the tests.

What Is Init(args)?

Init(args) is a seamlessly integrated and type safe framework for providing your Components and ScriptableObjects with their dependencies.

Main Features

- Add Component with arguments.
- Instantiate with arguments.
- Create Instance with arguments.
- new GameObject with arguments.
- Initializers – use interfaces, value providers.
- Services – automatically received by clients as Init arguments.
- Wrappers – attach plain old class objects to GameObjects.
- InitOnReset / InitInEditMode – auto-assign references in edit mode
- Type-safe and reflection-free dependency injection.

Introduction

Did you ever wish you could just call Add Component with arguments like this:

```
Player player = gameObject.AddComponent<Player,
IInputManager>(inputManager);
```

Or maybe you've sometimes wished you could Instantiate with arguments like so:

```
Player player = playerPrefab.Instantiate(inputManager);
```

And wouldn’t it be great if you could create ScriptableObject instances with arguments as well:

```
DialogueAsset dialogue = Create.Instance<DialogueAsset, Guid>(id);
```

This is precisely what Init(args) let’s you do! All you need to do is derive your class from one of the generic MonoBehaviour<T…> base classes, and you’ll be able to receive upto six arguments in your Init function.

In cases where you can’t derive from a base class you can also implement the Iinitializable<T> interface and manually handle receiving the arguments with a single line of code (see the InitArgs section of the documentation for more details).

Why Init(args)?

To fully understand what benefits Init(args) can unlock, one needs to first understand a key principle in software engineering: **inversion of control**.

What inversion of control means in a nutshell is that instead of classes independently retrieving specific objects to work with, they will work with what ever objects are provided for them by other classes. This comes with many benefits such as the ability to easily switch the objects passed to instances with different ones.

Usually one can achieve inversion of control by simply providing the objects that a class depends on in their constructor:

```
using UnityEngine;

public class Player
{
        public IInputManager InputManager { get; }
        public Camera Camera { get; }
        public Player(IInputManager inputManager, Camera camera)
        {
                InputManager = inputManager;
                Camera = camera;
        }
}
```

Unity’s MonoBehaviours or ScriptableObjects however can’t receive any arguments in this manner in their constructor, making it more difficult to use inversion of control.

This tends to lead to a situation where methods such as the Singleton pattern are used all over the place, tightly coupling classes with other classes in a tangled web of dependencies.

```
using UnityEngine;


public class Player : MonoBehaviour
{
        private void Update()
        {
                if(InputManager.Instance.Input.y > 0f)
                {
                        float speed = 0.2f;
                        float distance = Time.deltaTime * speed;
                        transform.Translate(Camera.main.transform.forward * distance)
                }
        }
}
```

While this does accomplish the job of retrieving the instance, it also comes with some pretty severe negative side effects that may end up hurting you in the long run, especially in larger projects:

- It can cause the dependencies of a class to be hidden, scattered around the body of the class, instead of all of them being neatly defined in one centralized place and tied to the creation of the object. This can leave you guessing about what prerequisitives need to be met before all the methods of a class can be safely called. So while you can always use GameObject.AddComponent<Player>() to create an instance of a Player, it's not apparent that an InputManager component and a main camera might also need to exist somewhere in the scene for things to work. This hidden web of dependencies can result in order of execution related bugs popping up as your project increases in size.
- It tends to make it close to impossible to write reliable unit tests. If the Player class depends on the InputManager class in specific, you can't swap it with a simple mock implementation that you'd be able to easily control during testing.
- Tightly coupling with specific classes can make it a major pain to refactor your code later. For example let's say you wanted to switch all classes in your code base from using the old InputManager to a different one like NewInputManager; you would need to go modify all classes that referenced the old class, which could potentially mean changing code in hundreds of classes. In contrast when using inversion of control, you might be able accomplish the same thing by changing a single line of code in your composition root (the place where the InputManager instance is created), and from there the new IInputManager gets forwarded to all other classes.

- Tight coupling with specific classes also means less potential for modularity. For example you can't as easily swap all your classes from using MobileInputManager on mobile platforms and PCInputManager on PC platforms. This limitation can lead to having bulky classes that handle a bunch of stuff instead of having lean modular classes that you can swap to fit the current situation.
- Tight coupling can also make it impossible to move classes from one project to another. Let's say you start working on a new game and want to copy over the Camera system you spent many months perfecting in your previous project. Well if your CameraController class references three other specific classes, and they all reference three other specific classes and so forth, that might have no choice but to start over from scratch.

The reason why Init(args) exists is to avoid all of these issues, by making it very easy to achieve inversion of control in a way that feels native to Unity.

If the Player class derives from the new generic MonoBehaviour base class, it will receive the inputManager and camera arguments in its **Init** function, where you'll be able to assign them to instance variables:

```
using UnityEngine;
using Sisus.Init;

public class Player : MonoBehaviour<IInputManager, Camera>
{
	public IInputManager InputManager { get; private set; }
	public Camera Camera { get; private set; }

	protected override void Init(IInputManager inputManager, Camera camera)
	{
		InputManager = inputManager;
		Camera = camera;
	}
}
```

MonoBehaviour<T...>

Init(args) contains new generic versions of the MonoBehaviour base class, extending it with the ability to receive up to twelve objects during initialization.

For example the following Player class depends on an object that implements the IInputManager interface and an object of type Camera:

```
public class Player : MonoBehaviour<IInputManager, Camera>
```

When you create a component that inherits from one of the generic MonoBehaviour base classes, you'll always also need to implement the **Init** function for receiving the initialization arguments.

```
public IInputManager InputManager { get; private set; }
public Camera Camera { get; private set; }

protected override void Init(InputManager inputManager, Camera camera)
{
   InputManager = inputManager;
   Camera = camera;
}
```

The Init function is called when the object is being initialized, before the Awake function.

Unlike the Awake function, Init always gets called even if the component exists on an inactive GameObject, so that the object will be able to receive its dependencies regardless of GameObject state.

OnAwake

Do not add an *Awake* function to classes that inherit from one of the generic MonoBehaviour classes, because the classes already define an Awake function. If you need to do something during the Awake event, override the *OnAwake* function instead.

OnReset

Do not add a Reset function to classes that inherit from one of the generic MonoBehaviour classes, because the classes already define a Reset function. If you need to do something during the Reset event, override the OnReset function instead.

Creating Instances

If you have a component of type Player that inherits from MonoBehaviour<IInputManager, Camera>, you can add the component to a GameObject and initialize it with an argument using the following syntax:

```
gameObject.AddComponent<Player, IInputManager, Camera>(inputManager, camera);
```

You can create a clone of a prefab that has the component attached to it using the following syntax:

```
prefab.Instantiate(inputManager, camera);
```

You can create a new GameObject and attach the component to it using the following syntax:

```
new GameObject<Player>().Init(inputManager, camera);
```

You can use the Inspector to specify the initialization arguments for your component by creating an Initializer for it and adding it to the same GameObject that contains your component.

```
public class PlayerInitializer : Initializer<Player, IInputManager, Camera> { }
```

In rare instances you might want to manually initialize an existing instance of the component without doing it through one of the pre-existing methods listed above. One example of a scenario where this might be useful is when using the Object Pool pattern to re-initialize pre-existing instances.

In order to manually call the Init function you must first cast the component instance to IInitializable<TArgument>.

```
var initializable = (Iinitializable<Targument>)component;

initializable.Init(argument);
```

Initialization Best Practices

It is generally recommended to only use the Init function to assign the received dependencies to variables, and then use OnAwake, OnEnable and Start for other initialization logic, such as calling other methods or starting coroutines.

There are a couple of different reasons for this recommendation:

- Init can get called in Edit Mode for example for any classes that have the InitOnResetAttribute. As such calling other methods from the Init function could result in unwanted modifications to being done to your scenes or prefabs in edit mode.
- Unlike Awake, OnEnable and Start, Init can also get executed even on components on inactive GameObjects. As such, starting coroutines from inside an Init method could fail in such a situation.
- If a component uses the constructor to receive its Init arguments, the Init function can get executed in a background thread. Since most of Unity's internal methods and properties are not thread safe, calling any of them from an Init function might be risky.

More Than Twelve Dependencies

Only a maximum of twelve arguments can be passed to MonoBehaviour<T…> objects. If you object requires more services than this, it is recommended to try and do some refactoring to reduce that number, for example, by splitting the class into two or more smaller classes.

If this is not feasible, you can pass more than twelve services to your objects by wrapping multiple services inside one container object. You can define a custom class for your service container (recommended for better readability), or use a value tuple:

```
public class ThirteenNumbers : MonoBehaviour<(int a, int b, int c, int d, int e, int f, int g, int h, int i, int j, int k, int l, int m)>
{
        private int a, b, c, d, e, f, g, h, i, j, k, l, m;

        protected override void Init((int a, int b, int c, int d, int e, int f, int g, int h, int i, int j, int k, int l, int m) args)
        {
                a = args.a;
                b = args.b;
                c = args.c;
                d = args.d;
                e = args.e;
                f = args.f;
                g = args.g;
                h = args.h;
                i = args.i;
                j = args.j;
                k = args.k;
                l = args.l;
                m = args.m;
        }
}
```

Then when initializing your object, instead of passing all the dependencies as separate arguments, you just pass the single container object.

```
var numbers = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13);

new GameObject<ThirteenNumbers>().Init(numbers);
```

ScriptableObject<T...>

Init(args) contains new generic versions of the ScriptableObject base class, extending it with the ability to specify upto six objects that the class depends on.

For example the following GameEvent class depends on a string and a GameObject.

```
public class GameEvent : ScriptableObject<string, GameObject>
```

When you create a class that inherits from one of the generic ScriptableObject base classes, you'll always also need to implement the **Init** function for receiving the arguments.

```
protected override void Init(string id, GameObject target)
{
    Id = id;
    Target = target;
}
```

The Init function is called when the object is being initialized, before the Awake function.

Creating Instances

If you have a ScriptableObject of type TScriptableObject that inherits from ScriptableObject<TArgument>, you can create a new instance of the class and initialize it with an argument using the following syntax:

```
Create.Instance<TScriptableObject, TArgument>(argument);
```

You can create a clone of an existing instance using the following syntax:

```
sciptableObject.Instantiate(argument);
```

Awake Event

Do not add an Awake function to classes that inherit from one of the generic ScriptableObject classes, because the classes already define an Awake function. If you need to do something during the Awake event, override the OnAwake function instead.

Reset Event

Do not add a Reset function to classes that inherit from one of the generic ScriptableObject classes, because the classes already define a Reset function. If you need to do something during the Reset event, override the OnReset function instead.

StateMachineBehaviour<T...>

Init(args) contains new generic versions of the StateMachineBehaviour base class, extending it with the ability to receive up to six objects during initialization.

For example the following AttackBehaviour class depends on an object that implements the Iattackable interface and a ScriptableObject asset of type AttackBehaviourSettings:

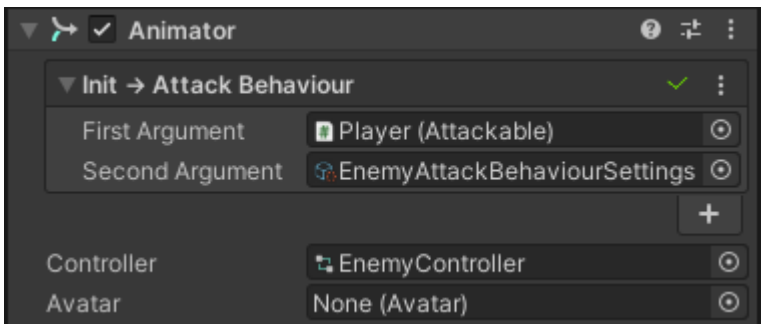
```
public class AttackBehaviour : StateMachineBehaviour<IAttackable,
AttackBehaviourSettings>
```

When you create a component that inherits from one of the generic StateMachineBehaviour base classes, you'll always also need to implement the **Init** function for receiving the initialization arguments.

```
public IAttackable Target { get; private set; }
private AttackBehaviourSettings settings;

protected override void Init(IAttackable target, AttackBehaviourSettings settings)
{
   Target = target;
   this.settings = settings;
}
```

State Machine Behaviour Initializers

You can use state machine behaviour initializers to specify the Init arguments that are used to initialize an instance of a StateMachineBehaviour<T…> derived class.



To achieve this do the following:

1. Create a state machine behaviour that derives from StateMachineBehaviour<T…>.
2. Create an AnimatorController asset.
3. Add your state machine behaviour to one or more states in the AnimatorController.
4. Add an Animator component to a GameObject.
5. Assign the AnimatorController asset to the Animator. After this you should see an Init section inside the Animator component in the Inspector.
6. Click on the + button in the Init section and select the option to generate an initializer for your state machine behaviour.
7. Once the initializer class has been generated use the Animator's Inspector to specify the initialization arguments.

Note that when initialization arguments are provided to a state machine behaviour using an initializer, the arguments will only get injected after the Awake and OnEnable event functions have already finished for the state machine behaviour. If you need to perform one-time setup for a state machine behaviour after they have received their Init arguments, you can either:

1.Execute your setup function at the beginning of the OnStateEnter function. Use a boolean variable to keep track of whether or not setup has been performed yet or not.

2.Execute your setup function at the end of the Init function.

OnAwake

Do not add an Awake function to classes that inherit from one of the generic StateMachineBehaviour classes, because the classes already define an Awake function. If you need to do something during the Awake event, override the OnAwake function instead.

OnReset

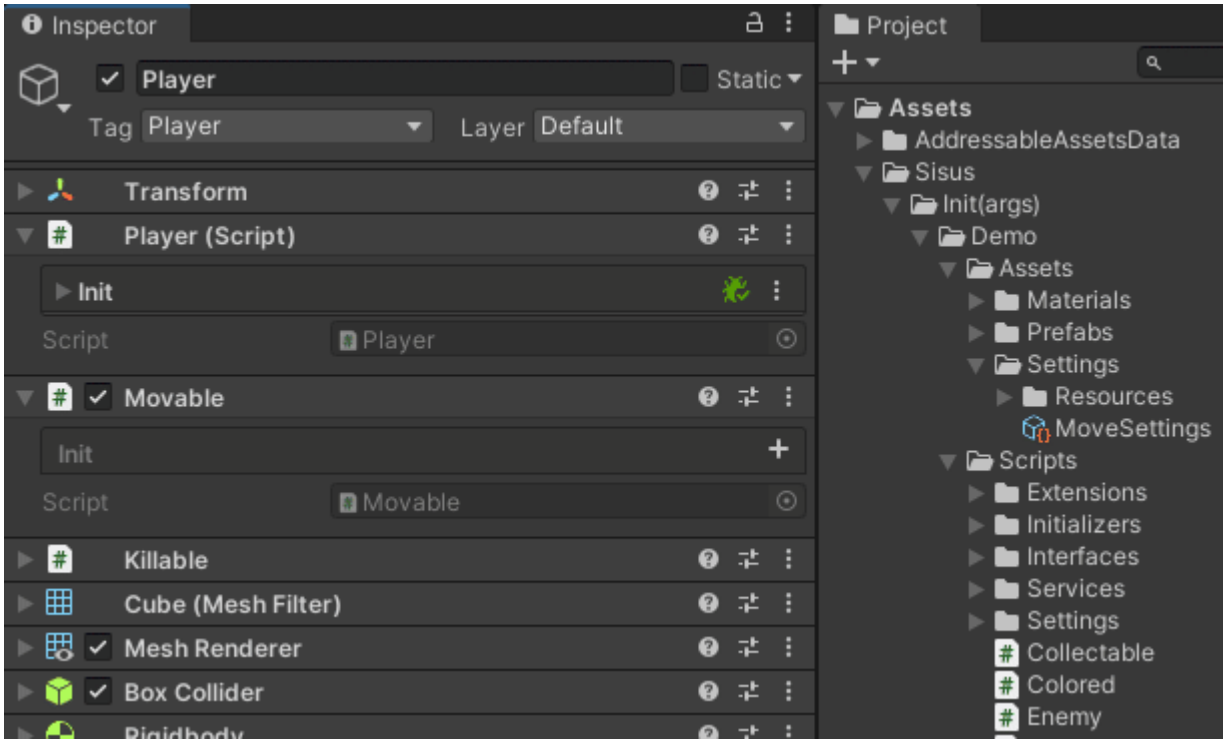
Do not add a Reset function to classes that inherit from one of the generic StateMachineBehaviour classes, because the classes already define a Reset function. If you need to do something during the Reset event, override the OnReset function instead.

Initializer

A major benefit of injecting dependencies to your classes through the Init method is that it makes it easy to decouple your components from specific implementations when you use interfaces instead of specific classes as your argument types. The list of arguments that the Init method accepts also makes it very clear what other objects the client objects depends on.

On the other hand, the ability to assign values using Unity's inspector is also a very convenient and powerful way to hook up dependencies; you can completely change object behaviour without having to write a single line of code!

The Initializer system is a solution that aims to marry the best of both worlds.

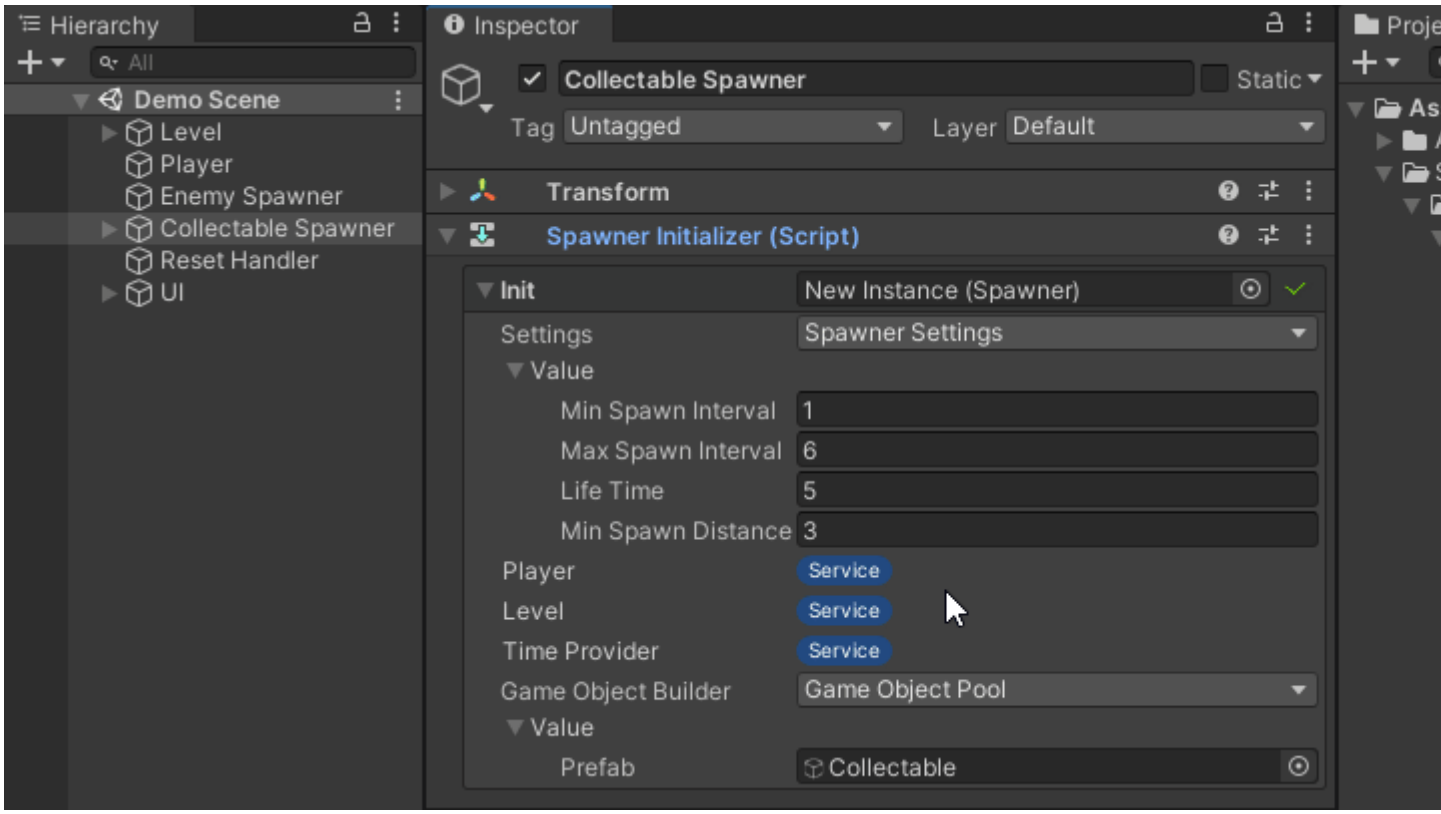


Benefits Of Using Initializers

- **Clearly Defined Dependencies** – You can see all dependencies of a component easily on both the code side and the Inspector side, just by looking at its Init arguments; no need to open the script and read through the whole thing to find all potential singleton references, GetComponent calls and serialized fields.
- **Interface Support** – With Initializers interfaces become first class citizens: you can drag and drop to assign interface type arguments, and they are serialized automatically as well (even if they derive from UnityEngine.Object).
- **Automatic Service Injection** – All Service dependencies are resolved automatically, so you don't need to drag-and-drop the same manager object to dozens of fields. This also means that you can easily change the service later on to use a different implementation, and all references to the service across the project update automatically!
- **Manual Service Overriding** – When the need arises, you can also change a single component to use something else besides the shared Service, simply by dragging-and-dropping another Object into the Init argument field. This gives you the perfect combination of making it easy to change a service project wide, while also giving you the flexibility to change the service on just a single client.
- **Easily Locate Services** – You can click the Service Tag on any service arguments in the Initializer's inspector to locate the service in question wherever it's located in the scene hierarchy or defined in a script asset.
- **Cross-Scene References** – Simply drag-and-drop a reference from another scene into an Init argument field, and it will persist even after the scenes are unloaded. Multi-scene workflows have never been easier.
- **Default Object Field Values** – You can add the InitOnReset attribute on the Initializer to automate the component setup process without having to clutter the client component with these implementation details.
- **Edit Mode Null Guard** – Initializers can automatically warn you about any missing references in edit mode.
- **Runtime Null Guard –** Initializers can automatically throw an exception when a missing reference is detected at runtime as well.
- **Unit Testable By Default** – When you use initializers, it also means that your components will have an Init function that the initializers can use to pass the arguments. This means that creating unity tests for your components becomes automatically so much easier, because you can initialize all your components using a single line of code, and substituting dependencies with mocks becomes trivial.
- **Separated Serialization Logic** – Since the Init arguments are serialized separately from the main component, you can freely use auto-implemented properties and any types you want, without having to worry about whether or not Unity can serialize them. You can finally work with dictionaries, tuples etc. without having to clutter your main component with serialization related boilerplate code. It also makes it possible to make your

collections and other fields read-only, so you can know that it's impossible for them to ever cause NullReferenceExceptions.

- **Single Responsibility Principle** – When the responsibility of resolving all dependencies is off-loaded to Initializers, it simplifies your main components and lets them focus solely on their main responsibilities. This makes it possible to better follow the single-responsibility principle and make your code more readable.



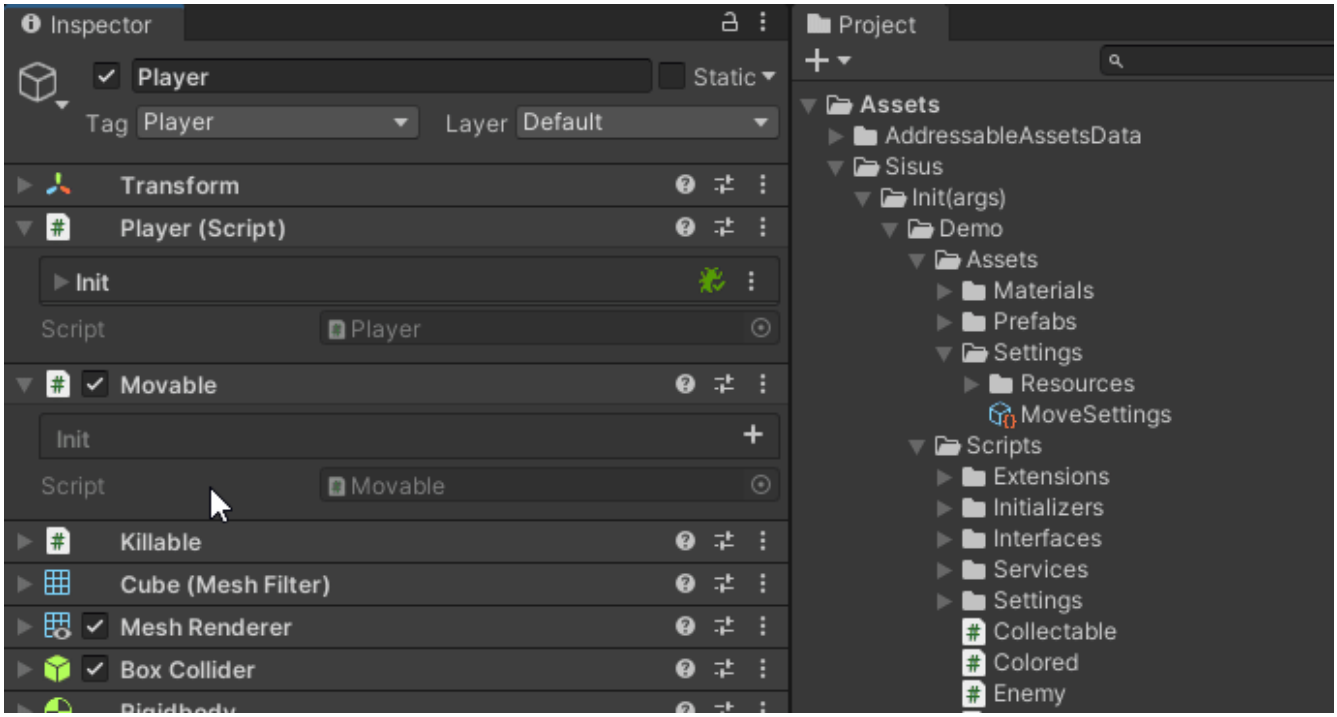
Creating Initializers

The Init section

The easiest and recommended way to create an Initializer for a component is to have Init(args) generate it for you automatically.

When you have a component that derives from MonoBehaviour<T…> or implements Iinitializable<T…>, an Init section will automatically appear at the top of the components of that type in the Inspector.

To generate an Initializer for the the component class, click on the + button in the Init section and select “Generate Initializer”.



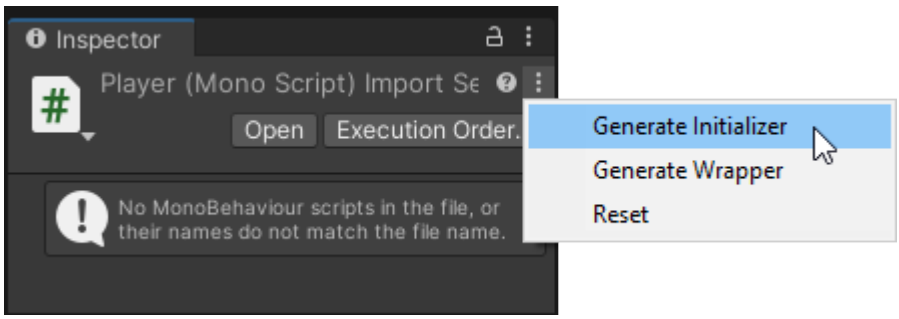
This will cause a new Initializer class to get automatically generated for your component class. It will be saved at the same location where your component script is located and named the same as your component class but with the “Initializer” suffix added.

When your component class already has an Initializer generated, you can use this same + button to attach an instance of the Initializer class to your component instead.

Script Asset Context Menu

In some case you might want to generate an Initializer for a class but it either is not a component type, or doesn’t have an Init section in the Inspector yet.
For example you might want to generate an Initializer for a plain old class object which you will be attaching to a GameObject using a Wrappers.

In such cases you can select the script asset that defines the class in the Project view and select “Generate Initializer” from the context menu.



Creating Manually With Code

Initializer<T…>

It is also easy to create new Initializers in code for your components by deriving from the Initializer<T…> base classes with the type of the component class as the first generic argument, followed by the types of its Init parameters.

For example, to define an Initializer for component Player, which derives from MonoBehaviour<IInputManager, Camera>, you would write the following:

```
public class PlayerInitializer : Initializer<Player, IInputManager, Camera> { }
```

InitializerBase<T…>

Initializers that derive from Initializer<T…> can hold and serialize Init arguments of UnityEngine.Object, types, as well as any types that Unity can serialize with the SerializeReference attribute. This includes interface types, and even in cases where the Init parameter is an interface type and the assigned value is a reference to a UnityEngine.Object type object.

This means that Initializer<T…> classes have all the same limitations that fields with the SerializeReference attribute have when it comes to serialization: most notably the lack of support for serializing generic types (with the exception of List<T>).

In cases where you need an Initializer to serialize an object that isn't supported by SerializeReference, you can derive from InitializerBase<T…>, implement the properties for all the arguments, and handle serializing them however you want.

```
using System;
using System.Collections.Generic;
using System.Linq;
using Sisus.Init;
using UnityEngine;
using Object = UnityEngine.Object;


public class DatabaseInitializer : InitializerBase<Database, Dictionary<string,
Object>>, ISerializationCallbackReceiver
{
    protected override Dictionary<string, Object> Argument { get; set; }


    [SerializeField]
    private SerializedElement[] serializedElements;


    void ISerializationCallbackReceiver.OnBeforeSerialize()
    {
        serializedElements = Argument.Select(element => new
SerializedElement(element.Key, element.Value)).ToArray();
    }


    void ISerializationCallbackReceiver.OnAfterDeserialize()
    {
        Argument = new Dictionary<string, Object>();
        Array.ForEach(serializedElements, (element) => Argument.Add(element.key,
element.value));
    }


    [Serializable]
    private sealed class SerializedElement
    {
        public string key;
        public Object value;


        public SerializedElement(string key, Object value)
        {
            this.key = key;
            this.value = value;
        }
```

```
    }
}
```

WrapperInitializer<T…>

To define an Initializer for a Wrapper, derive from the WrapperInitializer<T…> base class, with the type of the wrapped C# class as the first generic argument, followed by the types of its constructor parameters (assuming that a constructor is being used to pass in the object's dependencies).

Secondly you need to override the CreateWrappedObject function with parameters matching the constructor parameters of the wrapped class, and implement the logic for creating the wrapped object.

```
public class PlayerInitializer : WrapperInitializer<PlayerComponent, Player, IInputManager, Camera>
{
    protected override Player CreateWrappedObject(IInputManager inputManager, Camera camera)
    {
        return new Player(inputManager, camera);
    }
}
```

Circular Dependencies

Note that when using constructors to initialize your wrapped objects, it's possible to run into an issue with circular dependencies. For example, if Player's constructor requires an InputManager argument, and InputManager's constructor requires a Player argument, then it's not possible to create either object.

This issue can be resolved by splitting creation of the wrapped object into two phases: first getting/creating the instance, and secondly passing to it its Init arguments.

Get Or Create Instance Step

To enable the first step to happen, you need to also override the GetOrCreateUnitializedWrappedObject function.

```
public class PlayerInitializer : WrapperInitializer<PlayerComponent, Player, IInputManager, Camera>
{
    protected override Player GetOrCreateUnitializedWrappedObject()
    {
        return new Player();
    }

    protected override Player CreateWrappedObject(IInputManager inputManager, Camera camera)
    {
        return new Player(inputManager, camera);
    }
}
```

Alternatively if you add the [Serializable] attribute to the wrapped class, then the Wrapper will hold an uninitialized instance of it by default, from which the default implementation of GetOrCreateUnitializedWrappedObject can automatically retrieve it.

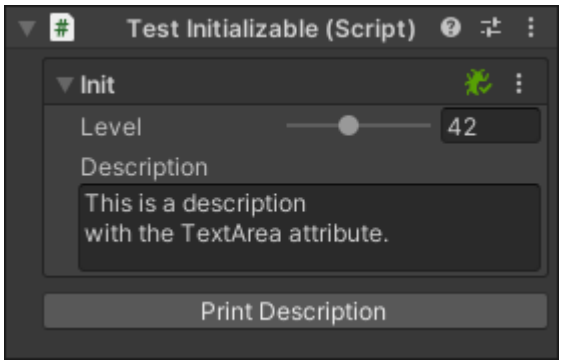
Init Step

To enable the second step to happen you need to have your wrapped class implement Iinitializable<T…> for receiving the initialization arguments in a deferred manner.

```
public class Player : IInitializable<IInputManager, Camera>

{

    private IInputManager inputManager;

    private Camera camera;


    public void Init(IInputManager inputManager, Camera camera)

    {

        this.inputManager = inputManager;

        this.camera = camera;

    }

}
```

With this two changes Init(args) can get past the circular reference by first creating the Player instance without initializing it with its Init arguments, then creating the InputManager instance with the Player instance passed to it, and then finally initializing the Player instance by passing in its Init arguments.

PropertyAttributes And Initializers



Sometimes you may want to customize how the Initializer arguments appear in the inspector by adding PropertyAttributes to them.

To do this, you will need to add a private nested Init class inside the Initializer, and then define a field for each Init argument accepted by the Initializer's client with their order and types matching those of the arguments in the client's Init method.

Any property attributes you attach to these fields will then get used when the corresponding initialization arguments are drawn in the Inspector.

```
public class PlayerInitializer : Initializer<Player, IInputManager, float>
{
    #if UNITY_EDITOR
    private class Init
    {
        public IInputManager inputManager;

        [Range(0f, 100)]
        public float speed;
    }
    #endif
}
```

Alternatively you can derive from InitializerBase<T…> which gives you the ability to manually define serialized fields to hold the data for all the Init arguments.

Any<Tvalue>

The Any<TValue> struct can be used to create serialized fields, into which you can assign any objects of the given type.

TValue can be of any type, including:

- Plain-old C# class
- UnityEngine.Object
- An interface type

You'll be able to select any assignable value for the serialized field using the Inspector, and it will be serialized for you.

Service Support

If a Service has the defining type T, then serialized fields of type Any<T > will automatically receive a reference to the service.

You are however still able to manually drag-and-drop some other Object into the field, to use that instead of the default Service instance.

Value Provider Support

Any value providers that can provide a value of type T can be drag-and-dropped into a serialized field of type Any<T >.

Example

If you wanted to make it possible to assign any object that implements the IInteractable interface using the Inspector, you could add a member field of type Any<IInteractable> to your component class:

```
[SerializeField]
Any<IInteractable> interactable;
```

You can then use the Value property to access the value stored inside the field.

```
void Interact()
{
    var interactable = this.interactable.Value;
    interactable.Interact(this);
}
```

ValueProviderMenu

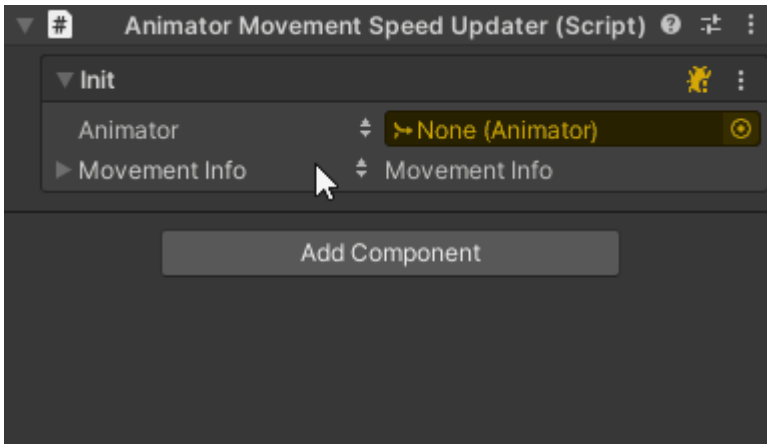
The ValueProviderMenu attribute can be used to introduce new menu items that will be shown in the dropdown menus for all Init arguments of targeted types.

The attribute can be added to a ScriptableObject-derived class that implements

IValueProvider<TValue> or IValueByTypeProvider.

You can then use the Value property to access the value stored inside the field.

Example



```
[CreateAssetMenu] // <- Create a single asset from the class
[ValueProviderMenu("Hierarchy/Get Component", Is.SceneObject)]
class GetComponent : ScriptableObject, IValueByTypeProvider
{
    public bool TryGetFor<TValue>(Component client, out TValue value)
    {
        if(client == null)
        {
            value = default;
            return false;
        }

        return Find.In(client.gameObject, out value);
    }
}
```

Shared Value Provider Asset

If you create a single asset from the value provider class somewhere in your project, then all Init arguments for which the menu item is selected, will be assigned a reference to that same single asset.

This can be useful for preserving memory, in cases where only a single value provider is needed.

Separate Value Providers

If you have zero, or more than one instances of the value provider class in your project, then each Init argument for which the menu item is selected, will be assigned its own instance of the value provider object.

This can be useful when you want to add serialized fields to the value provider, which each client can then use to configure the value provider to fit their individual needs.

Is Constraints

The following Is constraints can be applied for the targets of value providers with the

ValueProviderMenu attribute:

Unconstrained

No constraints applied to the types of Init parameters targeted.

Class

Shown in dropdown menus of reference type Init parameters.

ValueType

Shown in dropdown menus of struct type Init parameters.

Concrete

Shown in dropdown menus of concrete type Init parameters.

Abstract

Shown in dropdown menus of abstract type Init parameters.

Interface

Shown in dropdown menus of interface type Init parameters.

BuiltIn

Shown in dropdown menus of Init parameters of built-in C# types, such as bool, int, float and string.

Component

Shown in dropdown menus of component type Init parameters.
This includes both all component-derived types, as well as any interface types which are implemented by at least one component-derived type.

WrappedObject

Shown in dropdown menus of parameters of plain old C# types that have a Wrapper class.

SceneObject

Shown in dropdown menus of Init parameters of types that can exist as part of a scene.
This includes all component-derived types, all plain old C# types that have a Wrapper class, all interface types that are implemented by at least one component-derived type, and the GameObject type.

Asset

Shown in dropdown menus of Init parameters of types that can exist as a standalone asset, or as part of a prefab.
This includes all Object-derived types, all plain old C# types that have a Wrapper class, all interface types that are implemented by at least one Object-derived type, and the GameObject type.

Collection

Shown in dropdown menus of Init parameters that implement IEnumerable, such as lists and arrays.

Service

Shown in dropdown menus of Init parameters whose type is the defining type of a Service (registered using the ServiceAttribute).

> Note that you can add multiple constraints to one ValueProviderMenu attribute using the | operator.

Type Constraints

You can also provide one or more Type arguments in the ValueProviderMenu attribute's constructor, to constrain it to only appear in the dropdown menus of Init parameters that match those types.

Not Type Constraints

You can also assign a Type to the ***Not*** property in the ValueProviderMenu attribute's constructor, to constrain it to NOT appear in the dropdown menus of Init parameters whose type matches that type.

If you would like to stop it from appearing in the dropdown menus of more than one types of Init parameters, then you can assign an array of types to the ***NotAny*** property instead.

InitOnReset

When a component that derives from MonoBehaviour<T> and has the InitOnResetAttribute is first attached to a GameObject in the editor, or when the user hits the Reset button in the Inspector's context menu, the arguments accepted by the component are automatically gathered and passed to its Init function.

This auto-initialization behaviour only occurs in edit mode during the Reset event and is meant to make it more convenient to add components without needing to assign all UnityEngine.Object references manually through the inspector.

For example, when the following Player component gets added to a GameObject in edit mode, its Init method gets automatically called with Collider AnimatorController components from the same GameObject (if they exist):

```
[InitOnReset]
public class Player : MonoBehaviour<Collider, AnimatorController>
{
    [SerializeField]
    private Collider collider;

    [SerializeField]
    private AnimatorController animatorController;

    protected override void Init(Collider collider,
                                 AnimatorController animatorController)
    {
        this.collider = collider;
        this.animatorController = animatorController;
    }
}
```

By default component argument values are retrieved using the following logic:

1. First the same GameObject is searched for a component of the required type.
2. Second all child GameObjects are searched for a component of the required type.
3. Third all parent GameObjects are searched for a component of the required type.
4. Lastly all GameObjects in the same scene are searched for a component of the required type.

For each required component type the first match that is found using this search order is retrieved and then all the results are injected to the Init method.

It is also possible to manually specify which GameObjects should be searched for each argument. You can for example use the GetOrAddComponent value to try searching for the component in the same GameObject and then automatically adding it to it if no existing instance is found.

```
[InitOnReset(From.Children, From.GetOrAddComponent, From.Scene)]
public class Player : MonoBehaviour<Collider, AnimatorController, Camera>
```

InitInEditMode

Add this attribute to a component to have the arguments accepted by it be automatically gathered and passed to its Init function in Edit Mode, whenever any objects in the same scene or prefab that contains the component are modified.

This attribute supports any components that implement IInitializable<T>, which includes all classes that derive from MonoBehaviour<T>.

Dependencies for the component are automatically retrieved from the scene hierarchy, relative to the GameObject that holds the component with the attribute.

You can specify from where Object arguments should be searched, or use the default search mode, which tries to pick a good search mode to use based on the type of the argument:

- **Transform or GameObject**: From.GameObject
- **Other Component or interface**: From.Children or From.Parent or From.SameScene
- **Other Object**: From.Assets
- **Collection of Component, interface, Transform or GameObject**: From.Children
    - For example Collider[] or IEnumerable<Transform>

The found arguments are passed to the component's IInitializable<>.Init function, where they can be assigned to serialized fields.

This behaviour only occurs in edit mode and is meant to make it more convenient to add components without needing to assign all Object references manually using the inspector.

Example

For example, the following Player component would get its Init method automatically called in Edit Mode with the nearest Collider and AnimatorController components found in the same scene or prefab that holds the component (if both are found):

```
[InitInEditMode]
public class Player : MonoBehaviour<Collider, AnimatorController>
{
    [SerializeField]
    private Collider collider;

    [SerializeField]
    private AnimatorController animatorController;

    protected override void Init(Collider collider,
                                 AnimatorController animatorController)
    {
        this.collider = collider;
        this.animatorController = animatorController;
    }
}
```

By default component argument values are retrieved using the following logic:

1. First the same GameObject is searched for a component of the required type.
2. Second all child GameObjects are searched for a component of the required type.
3. Third all parent GameObjects are searched for a component of the required type.
4. Lastly all GameObjects in the same scene are searched for a component of the required type.

For each required component type the first match that is found using this search order is retrieved and then all the results are injected to the Init method.

It is also possible to manually specify which GameObjects should be searched for each argument. You can for example use the GetOrAddComponent value to try searching for the component in the same GameObject and then automatically adding it to it if no existing instance is found.

```
[InitInEditMode(From.Children, From.GetOrAddComponent, From.Scene)]
public class Player : MonoBehaviour<Collider, AnimatorController, Camera>
```

Initializer Support

It is also possible to add this attribute to an Initializer class.

From Initializer to Client

In this case, by default, the Init arguments will be taken from the Initializer, and passed to its client's Init function. In other words, no arguments will be gathered from the scene or prefab hierarchy in this case, and no arguments will be passed to the Initializer.

From Hierarchy to Initializer & Client

If you specify from where arguments should be retrieved from, using any values other than From.Initializer, or From.Default, then the Init arguments will be located from the scene or prefab hierarchy, and passed to both the Initializer and its client.

Services

What Are Services?

Services are objects that provide services to one or more clients that depend on them.

Init(args) automatically caches a single instance of each service and makes it simple to share that globally across all clients.

Defining Services In Code

Service Attribute

To define a class as a service, simply add the Service attribute to it.

```
[Service]
public class InputManager
{
      public bool MoveLeft => Keyboard.current[KeyCode.LeftArrow].isPressed;
      public bool MoveRight => Keyboard.current[KeyCode.RightArrow].isPressed;
}
```

This results in a single instance of the InputManager class being automatically created and cached behind the scenes.

Services can be plain old class objects or derive from MonoBehaviour or ScriptableObject.

Dependencies Between Service Classes

Services registered using the Service attribute should always have a parameterless default constructor, so that it is possible for the framework to create an instance of it automatically.

If your services need to make use of other services to function, you can implement an IInitializable<T…> interface with its generic types matching the types of other services. The other services will get injected to the client service through its Init method as part of the service initialization process.

```
[Service]
public class PlayerManager : IInitializable<InputManager>
{
      private InputManager inputManager;
      public void Init(InputManager inputManager) => this.inputManager = inputManager;
}
```

Unity Events For Services

Your Services, even if they don't derive from MonoBehaviour, can still receive callbacks during select Unity events by implementing one of the following interfaces:

1. **IAwake** – Receive a callback to an Awake function immediately following the initialization of the service instance. The Init functions of all services that implement an IInitializable<T> interface are always executed before any Awake functions are.
2. **IOnEnable** – Receive a callback to an OnEnable function following the initialization of the service instance. The Awake functions of all services that implement IAwake are always executed before any OnEnable functions are.
3. **IStart** – Receive a callback to a Start function following the initialization of the service instance. The OnEnable functions of all services that implement IOnEnable are always executed before any Start functions are.
4. **IUpdate** – Receive callback during the Update event.
5. **IFixedUpdate** – Receive callback during the FixedUpdate event.
6. **ILateUpdate** – Receive callback during the LateUpdate event.
7. **IOnDisable** – Receive callback when the application is quitting or when exiting play mode in the editor.
8. **IOnDestroy** – Receive callback when the application is quitting or when exiting play mode in the editor.

Additionally if your service class implements the IDiposable interface then IDispose will be called for it when the application is quitting or when exiting play mode in the editor

Services and Interfaces

One big benefit with the service system is that it supports caching and receiving services using an interface as the defining type.

This makes it very easy to decouple your classes from relying on specific concrete classes. This has numerous benefits such as making it easier to create unit tests for your classes and making it much easier to swap your services with other ones midway through development.

To achieve this, first we need to define the interface that all clients objects can use to communicate with the service object.

```
public interface IInputManager
{
    bool MoveLeft { get; }
    bool MoveRight { get; }
}
```

Next let's return to our InputManager class and make it implement the interface, as well as specify in the Service attribute that this service should be accessed via the interface type.

```
[Service(typeof(IInputManager))]
public class InputManager : IInputManager
{
    public bool MoveLeft => Keyboard.current[KeyCode.LeftArrow].isPressed;
    public bool MoveRight => Keyboard.current[KeyCode.RightArrow].isPressed;
}
```

After this everything can work exactly the same on the client side, except now instead of referring to the InputManager directly, we use the IInputManager interface instead.

```
public class Player : MonoBehaviour<IInputManager>
{
    private IInputManager inputManager;

    protected override void Init(IInputManager inputManager)
    {
        this.inputManager = inputManager;
    }
}
```

Using interfaces can be a powerful way to decouple your client classes from relying on specific service classes.

This makes it really easy to swap your services with other ones whenever you want, such as during unit testing.

It also makes it trivial to dynamically select the active service from multiple options for example using define directives.

```
#if UNITY_STANDALONE
[Service(typeof(IInputManager))]
#endif
public class KeyboardInputManager : MonoBehaviour, IInputManager
{
    public bool MoveLeft => Keyboard.current[KeyCode.LeftArrow].isPressed;
    public bool MoveRight => Keyboard.current[KeyCode.RightArrow].isPressed;
}
#if !UNITY_STANDALONE
[Service(typeof(IInputManager))]
#endif
public class GamepadInputManager : MonoBehaviour : IInputManager
{
    public bool MoveLeft => Gamepad.current[GamepadButton.DpadLeft].isPressed;
    public bool MoveRight => Gamepad.current[GamepadButton.DpadRight].isPressed;

    private void Awake()
    {
        if(Application.isConsolePlatform)
        {
            Service.SetInstance<IInputManager>(this);
        }
    }
}
```

Lazy Initialization

It is also possible to delay the initialization of services until the moment they are first needed.

If you set LazyInit = true then an instance of the service will not be created when the game starts like usually, but this will be delayed to only take place when the first client requests an instance.

This makes it possible to use the ServiceAttribute with FindFromScene load method, even in cases where the service component doesn't exist in the first scene of the game when the game is loaded.

```
[Service(LazyInit = true, FindFromScene = true)]
public class UIManager : MonoBehaviour
{
    [SerializeField]
    private Panels[] panels;


    public Panel ShowPanel(string panelId)
    {
        return panels.Where(panel => panel.Id == panelId).Show();
    }
}
```

In games with lots of services, lazy initialization can also be used to reduce the initial loading time of the game by spreading loading to happen over a longer period of time.

Lazy initialization was implemented using the static constructor feature in C# which means it is guaranteed to only happen once per service type and to be fully thread safe.
It also means that subsequently clients can retrieve instances of a service through a simple auto-implemented property without any additional overhead being introduced with reference fetching always happening through something like a Lazy<T> wrapper.

Services and Thread Safety

Instances of services class that have the Service attribute are created during initialization of the game, before the first scene is loaded. After this process has finished the service framework will not make any changes to these service instances.

Because the service instances will remain unchanged throughout the applications lifetime by default, accessing services through Service<T>.Instance is a thread safe operation, provided that this only occurs after the initialization process has finished and that you don't manually change services instance at runtime in your code.

If however you manually change Service instances at runtime using the Service.SetInstance method, accessing the service from other threads using Service<T>.Instance is no longer a safe operation.

Defining Services with the Inspector

In addition to using the ServiceAttribute, it is also possible to define services without the need for any code, using just the Inspector.

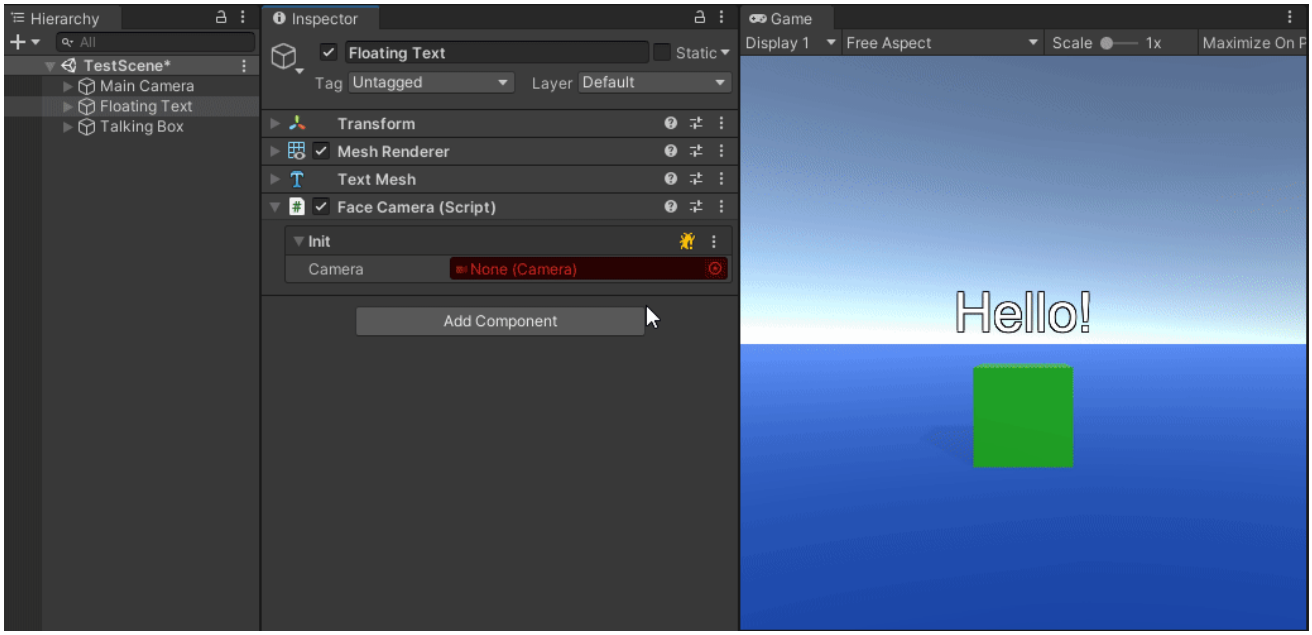
The methods can be useful for handling dynamic services that only exist in the scene hierarchy at particular times.

Think things like UI Panels that only become active when the user presses a button and become deactive again when the user closes them.

While all services registered using the ServiceAttribute are globally accessible to any clients, scene-based dynamic services can be limited to only be accessible to a limited subset of clients based on where they exist in the scene hierarchies relative to the GameObject that holds the service.

Service Tags

The simplest way to to define a single component as a service is to select the *Make Service Of Type… option* in the context menu for the component.
This opens up a dropdown menu that lets you select the defining type for the service – i.e. the type that clients can use to retrieve an instance of the service.



After selecting the defining type a blue Service Tag should appear in the header of the component in the Inspector. This acts as a marker to let you easily know which components are services, and also gives you access to some useful functionality when you left-click or right-click the Service Tag.

If you left-click the Service Tag the Defining Type menu will open again.
If you click the previously selected defining type of the service again, the service tag will be removed and the component will no longer be a service.
You can add more than one defining types for one component by selecting multiple options in the dropdown menu.

If you right-click the Service Tag a context menu will open, giving you access to additional commands.

Set Availability…

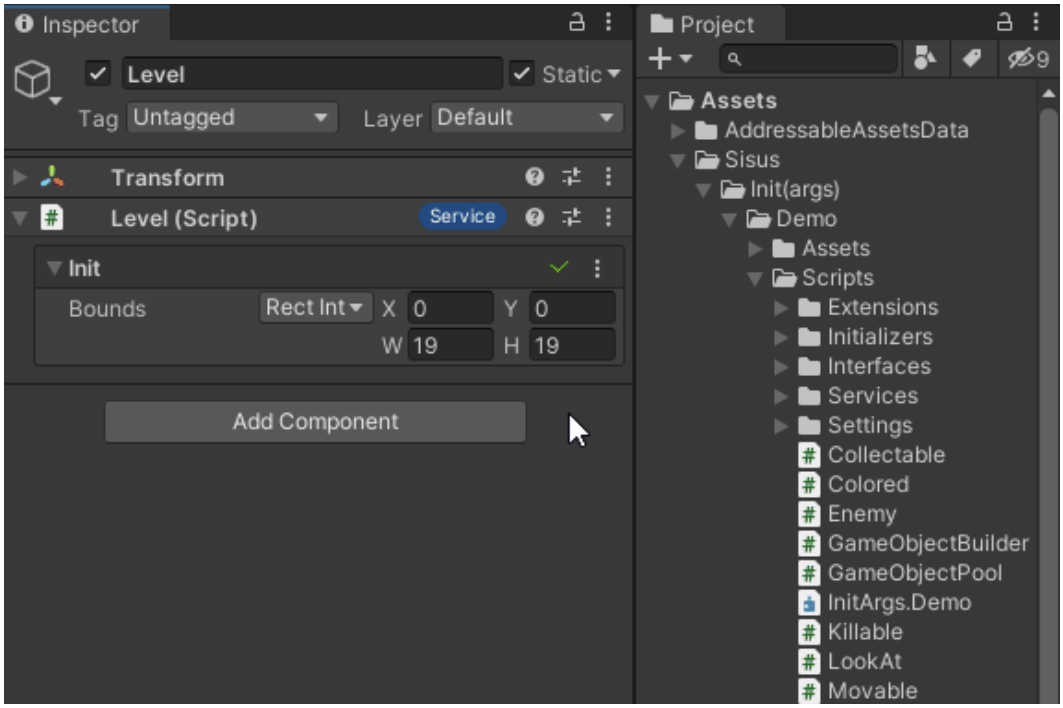
If you pick the **Set Availability…** menu item then a dropdown menu will open, letting you choose one of the following options to define which clients will have access to the service:

1. **In GameObject** – Only clients that are attached to the same GameObject as this component can receive services from it.
2. **In Children** – Only clients that are attached to the same GameObject as this component, or any of its children (including nested children), can receive services from it.
3. **In Parents** – Only clients that are attached to the same GameObject as this component, or any of its parents (including nested parents), can receive services from it.
4. **In Hierarchy Root Children** – Only clients that are attached to the GameObject which is at the root of this component's hierarchy, or any of the children of the root (including nested children), can receive services from it.
5. **In Scene** – Only clients belonging to the same scene as this component can receive services from it.
6. **In All Scenes** – All scene objects can receive services from this component, regardless of which active scene they belong to.
7. **Everywhere** – All clients can receive services from this component, regardless of of which active scene they belong to, or if they belong a scene at all.

Find Defining Object

Pick the **Find Defining Object** menu item to highlight the Object that defines the service in the Hierarchy or Project view.

If the service is defined by the ServiceAttribute, then the script asset that contains the attribute will be highlighted.

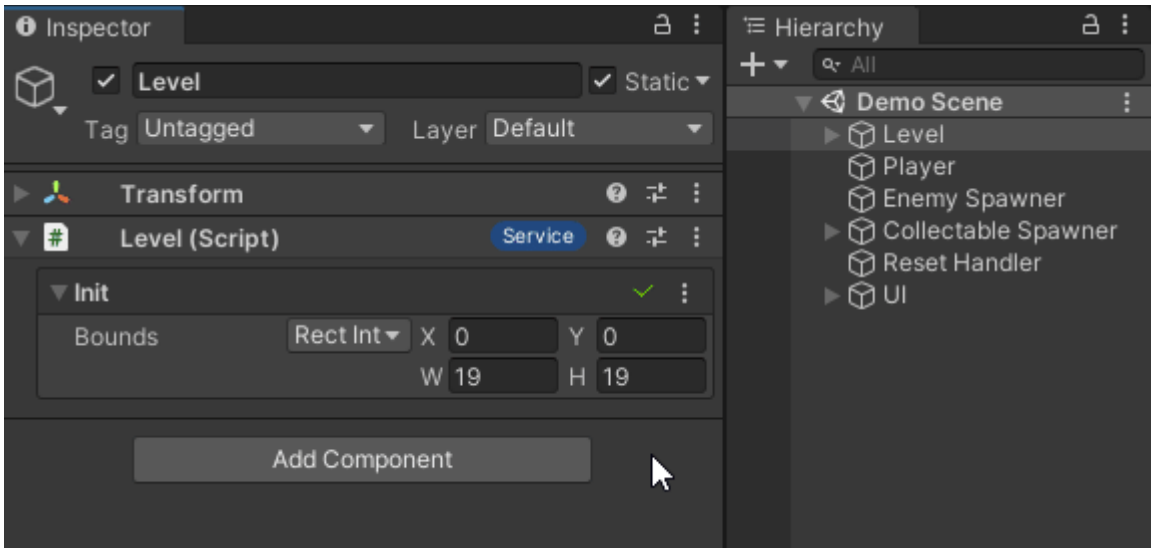


If the service is defined by a Services component, then the GameObject that contains the Services component will be highlighted.

If the service is defined by a Service Tag added via the *Make Service Of Type…* menu item then Find Defining Object will not appear in the menu.

Find Clients In Scenes

Pick the **Find Clients In Scenes** menu item to select all objects in the scene hierarchies that depend on the service.



Services Component

Another way to define scene services is through using the Services component.

Start by adding the Services component to a GameObject in the scene and adding a new entry to its **Provides Services** list.

Then drag the component you want to define as a Service to this list.
If you drag a GameObject to the list a popup will open to allow you to specify which component from that GameObject you want to define as a service.

After you've dragged a component to the list, press the button that appears on the right side of the Object field to open the defining type dropdown menu. Select the defining type for the service from the menu.

You can define as many services as you want using a single Services component.

You can only drag-and-drop components and scriptable objects to the Services Component, not plain old C# objects. If you want to register a plain old C# object as a service using a Services component, you can create a wrapper or a value provider, and drag-and-drop that into the Services Component.

Registering Services Manually In Code

In addition to registering services automatically using the Service attribute, it is possible to register them manually in code as well.

This might be useful for example if you want to dynamically swap your services with different ones based on some conditional logic.
For example you could swap the KeyboardInputManager to GamepadInputManager when the user changes their input method in the settings.

To manually register a service use the Service.SetInstance method.

To make sure that the service gets registered before the Awake method on any clients components or their initializers is executed, you can add the InitOrder attribute with Category set to ServiceInitializer to the component that does the registering.

```
[InitOrder(Category.ServiceInitializer)]

public class InputManager : MonoBehaviour

{

    void Awake()

    {

        Service.SetInstance<InputManager>(this);

    }

}
```

Using Services

To automatically receive the InputManager service during initialization you can have your class derive from MonoBehaviour<InputManager>. It will then receive the InputManager object in its Init function, making it possible to assign it to a member variable.

```
public class Player : MonoBehaviour<InputManager>

{

    private InputManager inputManager;


    protected override void Init(InputManager inputManager)

    {

        this.inputManager = inputManager;

    }

}
```

This means that the Player component could exist as part of a scene that is loaded or a prefab that is instantiated without any arguments being provided manually, and Init would still get called with the InputManager service.

Note that the Init function will only get automatically called during initialization if all the Init arguments the client expects are services.

If one or more arguments are not services, then you will need to manually provide all the arguments when initializing the client.

For example, consider this client that requires one service object and one object that is not a service:

```
public class Player : MonoBehaviour<InputManager, Camera>
{
    private InputManager inputManager;
    private Camera firstPersonCamera;

    protected override void Init(InputManager inputManager, Camera firstPersonCamera)
    {
        this.inputManager = inputManager;
        this.firstPersonCamera = firstPersonCamera;
    }
}
```

You can retrieve the cached instance of any service manually using Service<T>.Instance.

```
var inputManager = Service<InputManager>.Instance;
```

Then you need to pass that instance to the client when it is being created.

For example if you are creating the client by instantiating it from a prefab, you can use the Prefab.Instantiate function.

```
var inputManager = Service<InputManager>.Instance;

var firstPersonCamera = Camera.main;
var playerPrefab = Resources.Load<Player>("Player");
playerPrefab.Instantiate(inputManager, firstPersonCamera);
```

Or if the client is being built from scratch at runtime, you can use the GameObject.AddComponent function that accepts initialization arguments.

```
var playerGameObject = new GameObject("Player");
playerGameObject.AddComponent(inputManager, firstPersonCamera);
```

If the client is a scene object, you can generate an Initializer to provide the arguments to the client. When using Initializers all service arguments are passed to the client automatically, so you only need to assign the other arguments using the Inspector.

Reacting To Changing Services

In cases where your services might change throughout the lifetime of the application, you may want to make sure clients of the services are always using the most recently registered services.

To do this you can subscribe receive callbacks a service instance changes to a different, using the Service.AddInstanceChangedListener method.

When you do this you should also remember to unsubscribe using Service.RemoveInstanceChangedListener when the listener gets destroyed.

```
public class Player : MonoBehaviour<IInputManager>
{
    private IInputManager inputManager;

    protected override void Init(IInputManager inputManager)
    {
        this.inputManager = inputManager;
    }

    private void OnEnable()
    {
        Service.AddInstanceChangedListener(OnInputManagerInstanceChanged);
    }

    private void OnDisable()
    {
        Service.RemoveInstanceChangedListener(OnInputManagerInstanceChanged);
    }

    private void OnInputManagerInstanceChanged(IInputManager newInstance)
    {
        inputManager = newInstance;
    }
```

```
}
```

Wrapper

The Wrapper class is a component that acts as a simple wrapper for a plain old C# object.

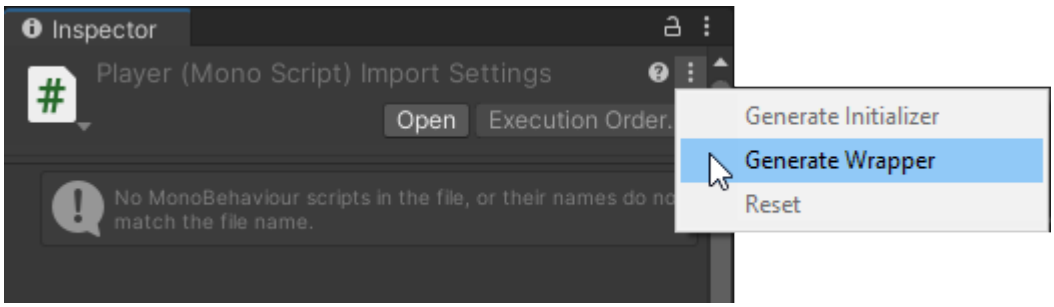
It makes it easy to take a plain old C# object and attach it to a GameObject and have it receive callbacks during any Unity events you care about such as Update or OnDestroy as well as to start coroutines running on the wrapper.

Creating a Wrapper

Let’s say you had a plain old C# class called Player:

```
using System;
using UnityEngine;

[Serializable]
public class Player
{
    [SerializeField]
    private Id id;

    [SerializeField, Range(0f, 10f)]
    private float speed;

    public Player(Id id, float speed)
    {
        this.id = id;
        this.speed = speed;
    }
}
```

The easiest way to create a Wrapper component for the class is to select the script, open its context menu in the Inspector and select **Generate Wrapper**.

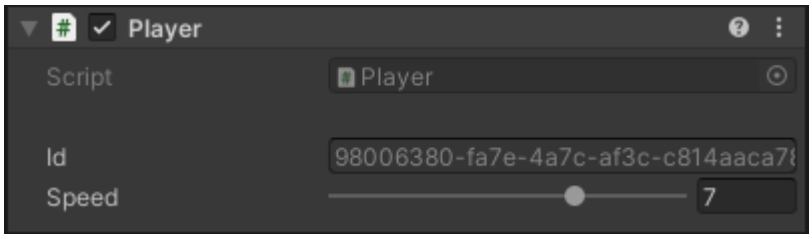


Or you could also create it manually by defining a class that inherits from Wrapper<Player>.

```
[AddComponentMenu("Wrapper/Player")]
class PlayerComponent : Wrapper<Player> { }
```

The AddComponentMenu attribute is optional; it just makes the PlayerComponent appear as simply “Player” in the Inspector and the Add Component menu.

If the wrapped class has the [Serializable] attribute, then an instance of it will be automatically created for any wrappers that exist as part of scene or prefab assets, and the objects serializable fields will be visible in the Inspector.



Initializing a Wrapper

SerializableAttribute

As stated before, if the wrapped class has the [Serializable] attribute, then an instance will be automatically created for all wrappers that exist as part of scene or prefab assets.

Provide Instance Via Constructor

If the wrapped object is not serializable, and doesn’t depend any external Object references or services, then you can initialize the instance right in the wrapper’s parameterless constructor, and pass it to the base constructor which accepts the instance as an argument.

```
[AddComponentMenu("Wrapper/Player")]
sealed class PlayerComponent : Wrapper<Player>
{
    public PlayerComponent() : base(new Player(Id.NewId(), 0f)) { }
}
```

Pass Instance During Instantiation

As wrappers implement IInitializable<TWrapped>, it means that you can also pass an instance to the component when creating an instance using Instantiate or AddComponent.

For example, an instance of Player could be attached to a game object like this:

```
Player player = new Player(id, 0f);
gameObject.AddComponent<PlayerComponent, Player>(player);
```

Wrapper Initializer

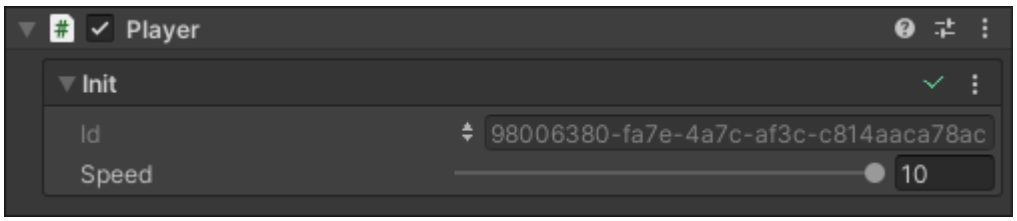
If the wrapped object depends on services, or you want to assign some arguments using the Inspector, you can generate an Initializer for the wrapper component.

The initializer should derive from WrapperInitializer<T…>, with its generic arguments of being the types of the wrapper component, the wrapped object, and all the objects the wrapped object needs to be provided to it.

```
public class PlayerInitializer : WrapperInitializer<PlayerComponent, Player, Id, float>
{
    private class Init
    {
        public Id Id;

        [Range(0f, 10f)] public float Speed;
    }

    protected override Player CreateWrappedObject(Id id, float speed)
    {
        Debug.Assert(id != Id.Empty);
        Debug.Assert(speed > 0f);

        return new Player(id, speed);
    }
}
```

You can optionally also define a nested class named “Init” inside the Initializer class, containing fields whose types match those of the arguments that the client accepts. If you do this, you can then attach property attributes to its fields, and their property drawers will be used when drawing the matching Init arguments in the Inspector.
When you use a Wrapper Initializer, it will take care of serializing all the arguments that will be passed to the client. This means that the client class itself does not necessarily need to be serializable.

```
public class Player
{
    public Id Id { get; }
    public float Speed { get; }

    public Player(Id id, float speed)
    {
        Id = id;
        Speed = speed;
    }
}
```



Unity Events

Wrapped objects can receive callbacks during select Unity events from their wrapper by implementing one of the following interfaces:

1. **IAwake** – Receive callback during the MonoBehaviour.Awake event.
2. **IOnEnable** – Receive callback during the MonoBehaviour.OnEnable event.
3. **IStart** – Receive callback during the MonoBehaviour.Start event.
4. **IUpdate** – Receive callback during the MonoBehaviour.Update event.
5. **IFixedUpdate** – Receive callback during the MonoBehaviour.FixedUpdate event.
6. **ILateUpdate** – Receive callback during the MonoBehaviour.LateUpdate event.
7. **IOnDisable** – Receive callback during the MonoBehaviour.OnDisable event.
8. **IOnDestroy** – Receive callback during the MonoBehaviour.OnDestroy event.

For example, to receive callbacks during the Update event the Player class would need to be modified like this:

```
public class Player : Iupdate
{
        public void Update(float deltaTime)
        {
                // Do something every frame
        }
}
```

Coroutines

Wrapped objects can also start and stop coroutines in their wrapper.

To gain this ability the wrapped object has to implement the ICoroutines interface.

```
public class Player : ICoroutines
{
	public ICoroutineRunner CoroutineRunner { get; set; }
}
```

The wrapper component gets automatically assigned to the CoroutineRunner property during its initialization phase.

You can start a coroutine from the wrapped object using CoroutineRunner.StartCoroutine.

```
public class Player : ICoroutines
{
	public ICoroutineRunner CoroutineRunner { get; set; }

	public void SayDelayed(string message)
	{
		CoroutineRunner.StartCoroutine(SayDelayedCoroutine(message));
	}

	IEnumerator SayDelayedCoroutine(string message)
	{
		yield return new WaitForSeconds(1f);

		Debug.Log(message);
	}
}
```

The started coroutine is tied to the lifetime of the GameObject just like it would be if you started the coroutine directly within a MonoBehaviour.

You can stop a coroutine that is running on the wrapper using CoroutineRunner.StopCoroutine or stop all coroutines that are running on it using CoroutineRunner.StopAllCoroutines.

```
public void OnDisable()
{
        CoroutineRunner.StopAllCoroutines();
}
```

Why Wrapped Objects?

The main benefit with using wrapped objects instead of a MonoBehaviour directly is that it can make unit testing the class easier.

You no longer subscribe to unity events via private magic functions hidden inside the body of the class, but have to explicitly implement an interface and expose a method for this. This makes it easier to invoke these functions in unit tests.

You also no longer need to think about scene management or creating GameObjects during unit testing which helps make it faster and easier to write reliable unit testing code.

Additionally the pattern that wrapped objects use to handle coroutines makes it easy to swap the coroutine runner class during unit tests, making it possible to even unit tests coroutines, for example with the help of the EditorCoroutineRunner class.

An additional benefit with using wrapped objects is that you can assign dependencies to read-only fields and properties in the constructor, which makes it possible to make your classes immutable. Having your wrapped objects be stateless can make your code less error-prone and makes it completely thread safe as well.

ScriptableWrapper

The ScriptableWrapper class is a ScriptableObject that can act as a simple wrapper for a plain old class object.

It makes it easy to take a plain old class object and serialize it as an asset in the project.

Let’s say you had a plain old class called Settings:

```
public class Settings
{
    public float MoveSpeed = 10f;
    public float JumpHeight = 4f;
}
```

To create a scriptable wrapper for the Settings class, create a class that inherits from ScriptableWrapper<Settings>.

```
public class SettingsAsset : ScriptableWrapper<Settings> { }
```

Beyond being able to wrap a plain old class object, this class functions just like a normal ScriptableObject, so you can decorate it with the CreateAssetMenu attribute to make it easy to create an instance of the asset from Unity’s create menu.

```
[CreateAssetMenu]
public class SettingsAsset : ScriptableWrapper<Settings> { }
```

You can access the plain old class wrapped by a ScriptableWrapper through the WrappedObject property.

```
var settingsAsset = Resources.Load<SettingsAsset>("Settings");
Settings settings = settingsAsset.WrappedObject;
```

This scriptable wrapper implements IInitializable<Settings>, which means that a new instance can be initialized with the Settings object passed as an argument using the custom Instantiate or Create.Instance methods that come with Init(args).

You can create create a new instance of the SettingsAsset with a Settings object wrapped inside of it using Create.Instance:

```
public void SaveSettings(Settings settings)
{
     var settingsAsset = Create.Instance<SettingsAsset>(settings);
     AssetDatabase.CreateAsset(settingsAsset, "Resources/Settings.asset");
}
```

You can clone an existing Settings asset while injecting a new Settings object inside of it using settingsAsset.Instantiate(settings);

```
public SettingsAsset InstantiateSettingsAsset(Settings overrideSettings = null)
{
     SettingsAsset settingsAsset = Resources.Load<SettingsAsset>("Settings");

     if(overrideSettings != null)
     {
          return settingsAsset.Instantiate(overrideSettings);
     }

     return Object.Instantiate(settingsAsset);
}
```

Unity Events

Wrapped objects can receive callbacks during select Unity events from their wrapper by implementing one of the following interfaces:

1. **IAwake** – Receive callback during the ScriptableObject.Awake event.
2. **IOnEnable** – Receive callback during the ScriptableObject.OnEnable event.
3. **IUpdate** – Receive callback during the Update event.
4. **IFixedUpdate** – Receive callback during the FixedUpdate event.
5. **ILateUpdate** – Receive callback during the LateUpdate event.
6. **IOnDisable** – Receive callback during the ScriptableObject.OnDisable event.
7. **IOnDestroy** – Receive callback during the ScriptableObject.OnDestroy event.

Find

The Find class is a utility class that helps in locating instances of objects from loaded scenes as well as assets from the project.

It has been designed from the ground up to work well with interface types, making it easier to decouple your code from specific concrete classes.

All methods in the Find class have the following features:

- Support for finding instances by interface type.
- Support for finding instances by the wrapped object type.
- Support for finding instances from inactive GameObjects.
- Support for TryGet style queries, where the method returns true if results were found or false if not.

The Find class includes the following static methods:

- **Find.Any** – Finds first instance of the provided type from active scenes.
- **Find.All** – Finds all instances of the provided type from active scense.
- **Find.InParents** – Finds object by type in GameObject and its parents.
- **Find.AllInParents** – Finds all objects by type in GameObject and its parents.
- **Find.InChildren** – Finds object by type in GameObject and its children.
- **Find.AllInChildren** – Finds all objects by type in GameObject and its children.
- **Find.WithTag** – Finds object with tag.
- **Find.GameObjectWith** – Finds first GameObject from active scenes that has an object of the given type attached to it.
- **Find.GameObjectOf** – Finds GameObject from active scenes that has the provided object attached to it.
- **Find.WrappedObject** – Finds plain old class object of the provided type wrapped by a rwapper component from active scenes.
- **Find.AllWrappedObjects** – Finds all plain old class objects of the provided type wrapped by wrapper components from active scenes.
- **Find.WrapperOf** – Finds wrapper component of the provided object.
- **Find.Wrapper** – Find first wrapper component from active scense that wraps a plain old class object of the provided type.
- **Find.AllWrappers** – Find all wrapper components from active scense that wrap a plain old class object of the provided type.
- **Find.In** – Finds first object of the provided type that is attached to a specific GameObject.
- **Find.AllIn** – Finds all objects of the provided type that are attached to a specific GameObject.
- **Find.Addressable** – Finds addressable asset synchronously. Has both runtime and edit mode support.
- **Find.Resource** – Finds object from the provided resources path.
- **Find.Asset** – Editor only method that finds asset of the provided type from the asset database.

GameObject<T…>

The generic GameObject structs are builders that can be used to initialize a GameObject and upto three components in a single line of code.

If you want a GameObject with a single component to be build, initialize a new instance of the GameObject<T> struct and specify the type of the component you want attached to the GameObject as the generic argument.

```
new GameObject<Camera>();
```

Optionally you can customize the state of the created GameObject, such as its name, parent, active state, position, rotation and scale.

```
new GameObject<Camera>("Camera", transform);
```

To finalize the building process you also need to call Init() on the GameObject<T> instance or cast it into a GameObject or the type of the created component.

```
Camera camera1 = new GameObject<Camera>("Camera 1").Init();
Camera camera2 = new GameObject<Camera>("Camera 2");
GameObject camera3 = new GameObject<Camera>("Camera 3");
```

If one of components being initialized derives from a MonoBehaviour<T…> base class or implements an IArgs<T…> interface, then it is also possible to pass dependencies to the component as arguments of the Init function.

For example, let's say we had we had the following IInputManager interface and class that implements the interface:

```
public interface IInputManager
{
    Vector2 Input { get; }
}

public class InputManager : IInputManager
{
    public Vector2 Input
    {
        get => new Vector2(Input.GetAxis("X"), Input.GetAxis("Y"));
    }
}
```

And let's say we had the following Player class that can be initialized with IInputManager and Collider arguments:

```
public sealed class Player : MonoBehaviour<IInputManager, Collider>
{
    public IInputManager InputManager { get; private set; }
    public Collider Collider { get; private set;}

    protected override Init(IInputManager inputManager, Collider collider)
    {
        InputManager =  inputManager;
        Collider =  collider;
    }
}
```

Then we could create a new instance of the Player component and initialize it using the following syntax:

```
IInputManager inputManager = new InputManager();

Player player = new GameObject<Player, BoxCollider>("Player")
                                    .Init1(inputManager, Second.Component);
```

The Init1 function tells the builder to initialize the first added component, i.e. the Player component, with the provided arguments.

The Second.Component token instructs the builder to pass the second added component, i.e. the BoxCollider, as the second initialization argument.

In this case you can skip calling the Init2 function of the builder, since only the first added component accepts initialization arguments.

If the generic arguments were reversed the syntax to build the same GameObject would look like this instead:

```
IInputManager inputManager = new InputManager();

Player player = new GameObject<BoxCollider, Player>("Player")
                                    .Init1()
                                    .Init2(inputManager, First.Component);
```

You always have to invoke all the Init functions in the same order as the component types have been defined in the generic arguments of the GameObject builder. Because of this you now also have to call the Init1 function to initialize the BoxCollider component, even though it does not accept any arguments.

Null / NullOrInactive

Init(args) has been designed with inversion of control in mind and aims to make it as easy as possible to work with interfaces instead of specific classes in Unity.

One pain point when using interfaces in Unity is that checking them for null can be problematic. This is because in addition to being actually null, Objects in Unity can also be destroyed. You usually don't even notice the difference when working with specific classes directly, because the Object class has a custom == operator which internally also checks whether the Object has been destroyed. However when an Object has been assigned to an interface type variable and this is compared against null, the overloaded == operator does not get used. This can result in null reference exceptions when trying to access the members of a destroyed Object.

```
Player player = FindAnyObjectByType<Player>();
IPlayer iplayer = player as IPlayer;

Destroy(player);

Debug.Log(player == null); // Prints ”True”
Debug.Log(iplayer == null); // Prints ”False”
```

To help with this problem in version a property has been added into every base class in Init(args): **Null**. When an interface type variable is compared against this Null property it functions just like the overloaded == operator in the Object class and returns true when the instance has been destroyed.

```
Player player = FindAnyObjectByType<Player>();
IPlayer iplayer = player as IPlayer;

Destroy(player);

Debug.Log(player == null);  // Prints ”True”
Debug.Log(iplayer == null); // Prints ”False”
Debug.Log(iplayer == Null); // Prints ”True”
```

It is also possible to utilize this safe null checking inside classes that don't derive from any of the base classes in Init(args) by importing the static members of the NullExtensions class.

```
using UnityEngine;

using static Sisus.NullExtensions;

public static class PlainOldClass
{
    public static void Example()
    {
        Player player = Object.FindAnyObjectByType<Player>();
        IPlayer iplayer = player as IPlayer;

        Object.Destroy(player);

        Debug.Log(player == null); // Prints true
        Debug.Log(iplayer == null); // Prints false
        Debug.Log(iplayer == Null); // Prints true
    }
}
```

In addition to the Null property there's also a new NullOrInactive property. This functions identically to the Null property but also checks if the Object is a component on an inactive GameObject.

```
Player player = FindAnyObjectByType<Player>();
IPlayer iplayer = player as IPlayer;

player.gameObject.SetActive(false);

Debug.Log(player.gameObject.activeInHierarchy); // Prints true
Debug.Log(iplayer == NullOrInactive); // Prints true
```

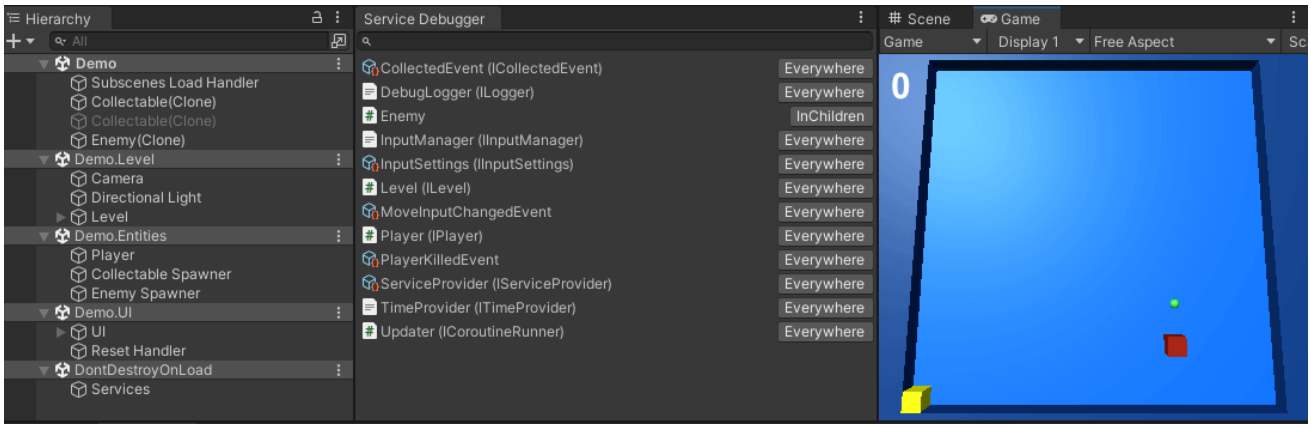
This functionality can be particularly useful when utilizing Object Pooling and an inactive GameObject is meant to represent an object which doesn't exist.

Service Debugger Window

Opening The Service Debugger Window

You can open the Service Debugger Window using the main menu item

Window > Analysis > Service Debugger.



Using The Service Debugger Window

The window lists all currently active services, which have been registered via ServiceAttributes, Service Tags and Services Components.

The first name shown in the list is the concrete type of the service object.

If the defining type of the service (the type which clients can use to retrieve the service) is not the same as the concrete type of the service, then that is listed after the concrete type inside parentheses.

On the right side you will see the availability of the service to potential clients. All services registered using the ServiceAttribute are available to clients ‘Everywhere’, while some services registered using Service Tags and Services Components can have more limited accessibility.

You can click the name of a service in the list, to locate the service in the Scene or Project hierarchy.

You can also click the client accessibility element to locate all clients of the service, currently located in the open scenes.

You can also right-click the name or the accessibility element to open a context menu with additional options.

IArgs<T…>

Classes that implement one of the generic IArgs<T…> interfaces can be provided with arguments during initialization (up to a maximum of twelve).

Methods such as Instantiate<TObject, T…> and AddComponent<TComponent, T…> can only be used to create instances of classes that implement one of the IArgs<T…> interfaces.

A contract to receive arguments

Any object that implements an IArgs<T…> interface makes a promise to receive arguments that have been injected for them during their initialization process using the InitArgs.TryGet method.

It is recommended that the arguments are retrieved during the Awake event function in most cases. This has the benefit of always only being executed once per object and occuring after the deserialization process has already finished.

Other possible options include the OnEnable and OnAfterDeserialize functions.

Technically it is also possible for MonoBehaviours to receive their dependencies in the constructor. However it is important to understand that the constructor gets called before the deserialization process, which means that the values of any serialized fields into which you assign your dependencies could get overridden during deserialization. If you only create your instances procedurally at runtime or only assign values to non-serialized fields, this can still be a workable solution, but beware that is is easy to make mistakes if you decide to go this route.

The Start event function is not considered to be part of the initialization process because it only gets executed during the next frame after the instance has been created.

If an Object that implements IArgs<T…> but does not receive the arguments that are passed to it an InitArgumentsNotReceivedException will be thrown. However, if the Object implements

IInitializable<T…> the arguments can be injected through the Init method, even if the client fails to receive the arguments independently, and in this case no exception will be thrown.

> **Note**: The Awake event function does not get called for components on inactive GameObjects. Because of this it is advisable for component classes to implement IInitializable<T…> and not just IArgs<T…> in most cases.

IInitializable<T...>

Methods

> void Init(TFirstArgument firstArgument, … TwelfthArgument twelfthArgument);

Classes that implement IInitializable<T…> have all the same functionality that they would get by implementing an IArgs<T…> interface, but with additional support for their **Init function** to be called manually by other classes.

This makes it possible for classes to inject dependencies even in cases where the object does not actively retrieve them during its initialization phase.

It also makes it possible to initialize the same object with dependencies more than once, which might be useful for example when utilizing an Object Pool to reuse your instances.

For MonoBehaviour derived classes it is generally recommended to implement IInitializable<T…> and not just IArgs<T…>. This is because the Awake event function does not get called for MonoBehaviour on inactive GameObjects, which means that dependency injection could fail unless the injector can manually pass the dependencies to it.

InitArgs

The InitArgs class is the bridge through which initialization arguments are passed from the classes that create instances to the objects that are being created (called clients).

Note that in most cases you don't need to use InitArgs directly; all of the various methods provided for initializing instances with arguments already handle this for you behind the scenes. But for those less common scenarios where you want to write your own code that makes use of the InitArgs you can find details about its inner workings below.

InitArgs.Set

The InitArgs.Set method can be used to store one to six arguments for a client of a specific type, which the client can subsequently receive during its initialization.

The client class must implement an IArgs<T…> interface with generic argument types matching the types of the parameters being passed in order for it to be targetable by the InitArgs.Set method.

```
InitArgs.Set<TClient, TArgument>(argument);
```

InitArgs.TryGet

The InitArgs.TryGet method can be used by a client object to retrieve arguments stored in InitArgs during its initialization phase.

```
if(InitArgs.TryGet(Context.Awake, this, out TArgument argument))
{
    Init(argument);
}
```

Use the Context argument to specify the method context from which InitArgs is being called, for example Context.Awake when calling during the Awake event or Context.Reset when calling during the Reset event.

InitArgs.Clear

Use InitArgs.Clear to release the arguments stored for a client from memory. This can be done by the class that provided the arguments for the client, after the client has been initialized.

If the stored arguments were never retrieved by the client, then InitArgs.Clear will return **true**. If this happens, it’s probably appropriate to log an error or throw an exception.

IvalueProvider<T>

Represents an object that can provide a value of type T on demand.

The value of a UnityEngine.Object-derived class that implements IValueProvider<T> can be assigned to an Init argument field of type T. When this is done, the value returned by the provider at runtime will be passed to the client during initialization.

This can enable patterns such as:

- ScriptableObject asset that provides a reference to a singleton instance which is only created at runtime.
- ScriptableObject asset that provides a dynamic primitive value such as an int that increments every second.
- A component that provides a dynamic primitive value such as a float that equals the distance between the Player GameObject and the GameObject that holds the component in question.
- ScriptableObject asset that returns a random position.