Documentation

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

Table of Contents

Installation....3
- Package Manager....3
- Keeping Up To Date....4
- Assembly Definitions....5
  - Recommended Assembly Setup....5
- Sisus.Inity Namespace....5
- Demo Scene....6
  - Installing Dependencies....6
  - Installing The Demo....6
  - Gameplay....6
  - Scene Overview....7

What Is Inity?....8
- Main Features....8
- Introduction....8

Why Inity?....9

MonoBehaviour<T...>....12
- OnAwake....12
- OnReset....12
- Creating Instances....12
- Initialization Best Practices....13

ScriptableObject<T...>....14
- Creating Instances....14
- Awake Event....14
- Reset Event....14

InitArgs....15
- InitArgs.Set....15
- InitArgs.TryGet....15
- InitArgs.Clear....15
- InitArgs.Received....15

Initializer....16

Any<T>....16

InitOnReset....17

Services....18
- Why Services?....18
- ServiceAttribute....19
- EditorServiceAttribute....20

Wrapper....21
- Unity Events....21
- Coroutines....22
- Why Wrapped Objects?....23

ScriptableWrapper....24
- Unity Events....24

Find....25

GameObject<T...>....27

Read-Only Members....29
- Using Reflection....29
- Using The Default Constructor....30
- Hybrid Solution....31

Interfaces....32
- IArgs<T...>....32
- IInitializable<T...>....32

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.

Assembly Definitions

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 Init(args) assemblies that contain the scripts you want to use.

Init(args) contains four assemblies that you can reference in your code:

1. **InitArgs** – The main assembly that contains most classes including MonoBehaviour<T>.
2. **InitArgs.Interfaces** – Assembly containing all interfaces such as IInitializable<T>.
3. **InitArgs.Services** – This contains only two attributes: ServiceAttribute and the EditorServiceAttribute. The reason that these attributes have been isolated to their own assembly is for performance reasons (more on that later).
4. **InitArgs.Editor** – Editor-only assembly containing classes related to unit testing as well as custom editors and property drawers for the inspector.

Recommended Assembly Setup

In most cases you will only need to use classes found in the *InitArgs* and in some cases the *InitArgs.Interface* assemblies, so add references to those to your assembly definition assets as needed. The assembly where you will have all your components that inherit from MonoBehaviour<T...> will need to reference at least the *InitArgs* assembly.

The *InitArgs.Editor* assembly you should only reference from your own Editor-only assemblies. This might be useful when writing 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 in edit mode tests.

For optimal performance it is recommended, if possible, to isolate all your service classes to their own assembly. If you manage to pull this off, then this assembly will be the only one that needs to reference the InitArgs.Services assembly. During initialization reflection will be used to scan all classes in assemblies that reference InitArgs.Services assembly, so the less classes there are to examine the faster this will be.

Since two-way references between assembly definition assets are not allowed, this will likely mean in practice that you can not **directly** references any classes contained in the assembly that houses your services. While this might seem like an insurmountable obstacle at first, it can also be a blessing in disguise, since it effectively forces you to **decouple** all your services from their actual implementations. You can pull this off by adding all your **interfaces into their own assembly**, having all your services implement one of these interfaces, and then interact with your services through these interfaces instead of referencing the classes directly.

You can take a look at the Demo project as an example of how to organize your scripts in the recommended manner.

Sisus.Init Namespace

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

```
using Sisus.Init;
```

Demo Scene

Init(args) comes with a demo scene containing a simple game built utilizing many of the features offered by the toolset.

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 Scene*. 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 scene contains six GameObjects in its hierarchy:

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, Killable and Movable components. The Player component is responsible for detecting collisions with ICollectable object and the Killable component with IDeadly objects. The Movable component moves the GameObject within the bounds of the level in response to user input.
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 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 Collectable.Collected UnityEvent is invoked and the Game Over text has been hooked to become active when the Player.Killed UnityEvent is invoked.

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.
- Service framework (a powerful alternative to Singletons).
- Wrapper system (attach plain old class objects to GameObjects).
- Auto-Initialization support.
- Assigning to read-only fields and properties.
- Type safety thanks to use of generics.
- 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 five 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 that Init(args) can offer, one needs to first understand a key principle in software engineering: **inversion of control**.

What inversion of control means is that instead of classes independently creating or retrieving references to other objects of specific types, they will work with whatever objects are provided for them by other classes.

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 hidden 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 so 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 prerequisites 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 good unit tests; if the Player class depends on the InputManager class in specific, you can't swap it with a mock implementation that you'd be able to easily control during testing.
- Tight 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 could be able to pull the same thing off by changing just 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 leave you with no choice but to start over from scratch every time.

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

When using Init(args) you no longer need to scatter your dependencies all over your classes, but can instead define all them neatly in a single and consistent place, in the definition of the class.

So, if your Player class requires an input manager and a camera to function, you can have your Player class derive from MonoBehaviour<IInputManager, Camera>.

Then you just implement the Init method to receive the inputManager and camera arguments from the class that creates the Player instance.

```
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;
    }

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

This small change comes with several major benefits:

- It becomes very apparent what other objects are needed for the Player class to function: an IInputManager and a Camera. Even if the Player class eventually grew to be 1000+ lines long, you could still find all the objects it depends on clearly listed in the class definition without having to read through the whole thing.
- The class no longer references the InputManager class directly, but instead communicates through the IInputManager interface. This makes it very easy to swap to using a different IInputManager implementation for example during unit testing.
- If you modify the Player class in the future and introduce a new dependency to it, you will instantly get compile errors from all existing classes that are trying to create an instance of the Player class without providing this necessary dependency. This makes it really easy to find all the places you need to go modify to ensure the Player object gets setup properly in all places in your code base.

MonoBehaviour<T...>

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

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 arguments.

```
protected override void Init(IInputManager 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 TComponent that inherits from MonoBehaviour<TArgument>, you can add the component to a GameObject and initialize it with an argument using the following syntax:

```
gameObject.AddComponent<TComponent, TArgument>(argument);
```

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

```
prefab.Instantiate(argument);
```

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

```
new GameObject<TComponent>().Init(argument);
```

You can add the component to a scene or a prefab and use Unity's inspector to specify the argument used to initialize the component by defining an Initializer for the component and adding one to the same GameObject.

```
public class TComponentInitializer : Initializer<TComponent, TArgument> { }
```

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 reuse existing instances.

To manually call the Init function first cast the component instance to IInitializable<TArgument> and then call the Init method on it.

```
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 or 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.
- Because Init can get called on inactive GameObjects it means that any calls to StartCoroutine will fail in this situation. As such starting coroutines is usually better to be done during the OnAwake event instead of in the Init function.
- 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.

ScriptableObject<T...>

Init(args) also contains new generic versions of the ScriptableObject base class, extending it with the ability to specify upto five 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.

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 Init(args) provides 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 documentation explaining its inner workings below.

InitArgs.Set

The InitArgs.Set method can be used to store one to five 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);
```

The generic TClient type does not have to be the exact type of the client but can also be its base class.

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 remove arguments cached in InitArgs. This should usually be called by the class responsible for providing the arguments to the client after initialization.

InitArgs.Received

Use this method to determine whether or not Init arguments provided for a client have been received yet or not.

This can be useful for throwing an exception if the client object has failed to receive the arguments during its initialization phase.

Initializer

A major benefit of injecting dependencies from the outside through the Init method is that it makes it easy to decouple your components from specific implementations, provided 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!

You can define an Initializer for component TComponent which inherits from MonoBehaviour<TArgument> using the following syntax:

```
TComponentInitializer : Initializer<TComponent, TArgument> { }
```

Then if you add both TComponent as well as TComponentInitializer to the same GameObject, you can specify all the arguments that TComponent's Init method takes using the inspector.

Initializers allow you to assign argument values even for interface types, and regardless of whether or not they derive from UnityEngine.Object or are plain old class objects.

Another cool benefit of the Initializer system is that it automatically detects when argument types are service types, and can automatically retrieve instances to the these at runtime. Thus you can easily mix and match service arguments and arguments that have been manually assigned through the inspector.

Any<T>

The Any<T> struct is similar to the Initializer class, but instead of it allowing you to specify all the Init arguments for a client, the Any<T> struct lets you specify just one argument.

For example, if you wanted to make it possible to assign any class that implements the IPlayer interface using the inspector, you could use the following syntax:

```
[SerializeField]
Any<IPlayer> player;
```

To get the value that was assigned to the Any struct simply call the Value property on it.

```
player.Value.MoveTo(position);
```

Similar to the Initializer class, Any<T> also supports automatically retrieving service instances at runtime.

InitOnReset

When a component that derives from MonoBehaviour<T> and has the InitOnResetAttribute is first added 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>
```

Services

Init(args) comes with a powerful Service framework that makes it easy to automate the creation and caching of instances of services which can then be injected to multiple services that depend on them.

Why Services?

If you are familiar with the Singleton pattern, the service system is a little bit similar, but has various benefits over it:

- Service instances can be retrieved using an interface they implement, which makes it possible to decouple your clients from relying on specific service classes. Swapping services for example during unit testing with mock implementations is trivial with the service system!
- Service instances can be automatically injected to Initializers and Any<T> fields.
- Services can use other services.
- All services are created automatically before the first scene is loaded, so they are always ready to be used even in Awake and OnAfterDeserialize regardless of script execution order settings.
- Retrieving service instances is fast and safe because there is no need to do it using a method that always does null-checking, thread locking and handles creating the instance lazily on-the-fly as needed.
- Services can be Components, ScriptableObjects or plain old class objects.
- No need to use a base class.

ServiceAttribute

To register a class as a service simply add the ServiceAttribute to it.

```
[Service]
public class InputManager
```

After this a shared instance of InputManager gets automatically created during initialization, and it can be retrieved by any clients that need it using the following syntax:

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

It is also possible (and highly recommended) to not register services using the type of the concrete class, but by the type of an interface that the class implements.

```
[Service(typeof(IInputManager))]
public class InputManager : IInputManager
```

With this change the shared instance of InputManager is retrieved instead using the IInputManager interface type, making it trivial to swap the InputManager with another class if needed:

```
IInputManager inputManager = Service<IInputManager>.Instance;
```

By default the shared service instance is created by initializing a new object. However when it comes to components, there are a couple of other options that can be used as well.

One option is to retrieve the Service component from the first loaded scene. To do this make sure the service inherits from the MonoBehaviour class and set FindFromScene as true in the Service attribute definition.

```
[Service(typeof(IInputManager), FindFromScene = true)]
public class InputManager : MonoBehaviour, IInputManager
```

This method of retrieving services is generally only recommended for projects that only have a single scene.

Another option is to retrieve the Service component from a prefab that is loaded from a Resources folder. To do this set ResourcePath to match the path of the asset within the Resources folder.

```
[Service(typeof(IInputManager), ResourcePath = ”Input Manager”)]
public class InputManager : MonoBehaviour, IInputManager
```

This method can also be used to load ScriptableObject assets in addition to prefabs.

If you have the Addressables package installed, it is also possible to load a service asset using its addressable key:

```
[Service(typeof(IInputManager), AddressableKey = ”Input Manager”)]
public class InputManager : MonoBehaviour, IInputManager
```

EditorServiceAttribute

The editor service attribute is identical to the Service attribute in function, except it is used to define services that are usable in the editor in edit mode instead of at runtime.

To register a class as an editor service add the EditorService attribute to it.

```
[EditorService(typeof(ILogger))]
public class Logger : ILogger
```

After this the shared instance can be retrieved using the Service class in edit mode, just like you would retrieve classes registered using the ServiceAttribute at runtime:

```
ILogger logger = Service<ILogger>.Instance;
```

Note that possible to add both the ServiceAttribute and the EditorService attribute to the same class, making it possible to use the same service both at runtime as well as in edit mode.

```
[Service(typeof(ILogger)), EditorService(typeof(ILogger))]
public class Logger : ILogger
```

Wrapper

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

It makes it easy to take a plain old class 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.

Let's say you had a plain old class called Player:

```
public class Player { }
```

To create a wrapper component for the Player class, make a class that inherits from Wrapper<Player>.

```
public class PlayerComponent : Wrapper<Player> { }
```

This wrapper implements IInitializable<Player>, which means that a new instance can be initialized with the Player object passed as an argument using any of the various methods listed in the Creating Instances section.

For example, the Player can be attached to a GameObject using the following syntax:

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

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, serialize it as an asset in the project.

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

```
public class Settings { }
```

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

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

This wrapper implements IInitializable<Settings>, which means that a new instance can be initialized with the Player object passed as an argument using any of the various methods listed in the Creating Instances section.

For example, the Player can be attached to a GameObject using the following syntax:

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

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 wrapper 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 builder 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.

Read-Only Members

Sometimes you may want to make use of read-only fields or get-only properties in your components and scriptable objects. Read-only members make your data immutable and as such result in code that is less prone to errors and fully thread safe without needing any complicated thread locking.

The issue is that you can't pass any constructor arguments to components or scriptable objects when creating them in Unity by default.

Init(args) offers three different solutions for this issue:

1. Define the read-only members in a and use a wrapper component to attach it to GameObject.
2. Assign arguments passed to the Init function to read-only members using reflection.
3. Retrieve initialization arguments inside the parameterless constructor via InitArgs.

For more information about using plain old class objects to achieve this refer to the documentation for the Wrapper<T> class.

Using Reflection

The MonoBehaviour<T…> and ScriptableObject<T…> base classes contain custom indexers that allow you to assign arguments that have been passed to your Init function into fields or properties even if they are read-only. This is possible thanks to the usage of reflection behind the scenes.

The syntax for assigning to read-only members inside the Init function looks like this:

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

    protected override void Init(IInputManager inputManager)
    {
        this[nameof(inputManager)] = inputManager;
    }
}
```

Note that there is a performance cost to using this method, because setting values using reflection is something like 10 times slower than assigning them directly. However, for classes that don't get initialized that often during gameplay, for example only during initial loading, this cost in performance isn't that bad and most likely won't even be noticeable in practice.

Using The Default Constructor

Another way to assign to read-only members with Init(args) is to manually receive initialization arguments in the parameterless constructor and assign them to read-only members there.

To achieve this make sure your class implements an IArgs<T...> interface with generic argument types matching the types of the arguments that the class can receive, and then use InitArgs.TryGet to retrieve arguments inside the constructor.

If your class can not function without receiving these arguments, then it is also advisable to log an error or throw an exception in the constructor if no arguments have been injected for it.

```
public class Player : MonoBehaviour, IArgs<IInputManager>
{
    public readonly IInputManager inputManager;

    public Player()
    {
        if(InitArgs.TryGet(Context.Constructor, this, out inputManager))
        {
            throw new MissingInitArgumentsException(this);
        }
    }
}
```

While this method of assigning to read-only members works without requiring any reflection, there are some big limitations to it as well (there is a reason why the MonoBehaviour<T…> classes use Awake instead of the constructor to receive arguments by default).

1. The constructor gets invoked before the deserialization process takes place. This means that if you assign arguments to any fields that are serialized, those values will be overridden during deserialization of scene objects and objects instantiated from prefabs. This issue can be avoided by building your components at runtime from scratch using AddComponent or new GameObject<T…>, or by taking care to only assign values to non-serialized members.
2. Constructors for scene objects are called before the Awake methods of any other objects in the same scene or instantiated prefab, regardless of Script Execution Order settings. This means, for one, that Initializers are not able to inject arguments for clients in the same scene before the constructors are already executed.
3. The constructor gets called for scene objects before a scene has finished fully loading. In the very first scene of the game constructors can even get called before static methods that have the RuntimeInitializeOnLoadMethod attribute - which is what is used to initialize all services in Init(args)! This means that if you have any components using constructor injection in the very first scene that is loaded when running the game, you might not be able to retrieve any services during their initialization.

For the above reasons it is not recommended to use this pattern at all with scene objects, and even with prefabs that are instantiated you have to be careful to avoid assigning arguments into serialized fields. On the other hand for components that you know will be added to GameObjects at runtime this pattern can work really well.

Hybrid Solution

Note that you don’t necessarily have to pick between using reflection or the default constructor for assigning arguments to read-only members: there is nothing stopping you from supporting both options.

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

        public Player()
        {
                InitArgs.TryGet(Context.Constructor, this, out inputManager);
        }

        protected override void Init(IInputManager inputManager)
        {
                this[nameof(inputManager)] = inputManager;
        }
}
```

In the above example the Player class will receive intialization arguments manually inside its parameterless constructor, if they have been provided already at this point. This option will get used when the object is created procedurally at runtime using AddComponent, Instantiate or new GameObject<T...>.

In addition to this the class also supports external classes manually invoking its Init function. This makes it possible for Initializers to also target instances of this class in scenes and prefabs.

With this approach you can get maximal performance whenever possible without making sacrifices on the reliability front.

Interfaces

IArgs<T...>

Classes that implement one of the generic IArgs<T…> interfaces can receive one or more arguments, up to a maximum of five, during initialization.

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

Classes that derive from MonoBehavour or ScriptableObject should usually retrieve their dependencies during the Awake event function. One thing to note though is that the Awake event function does not get called for components on inactive GameObjects, so it is recommended for all MonoBehaviour-derived classes to implement IInitializable<T…> instead for more reliable dependency injection.

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.

IInitializable<T...>

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 can’t receive them independently during initialization.

It also makes it possible to initialize objects with dependencies more than once, which might be useful for example when utilizing the Object Pool pattern 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.