Documentation

Table of Contents

Installation

Package Manager

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

Keeping Up To Date

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

The process for updating Init(args) Lite 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) Lite** 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) Lite is now updated to the latest version.

Assembly Definitions

To reference Init(args) Lite's code in your code, you need to create assembly definition assets in the roots of your script folders, and add references to the **InitArgs** assembly.

Sisus.Init Namespace

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

```
using Sisus.Init;
```

Sisus.NullExtensions

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;
```

What Is Init(args) Lite?

Init(args) extends the MonoBehaviour base class with the ability to pass arguments during initialization.

Main Features

- Add Component with upto twelve arguments.
- Instantiate with upto twelve arguments.
- Arguments received at the beginning of the Awake event
- Type-safe
- Reflection-free

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);
```

This is precisely what Init(args) Lite 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 twelve 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).

MonoBehaviour<T...>

Init(args) Lite contains new generic versions of the MonoBehaviour base class, extending it with the ability to specify upto twelve 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, at the beginning of the Awake event.

Unlike the Awake function, Init gets called even if the component exists on an inactive GameObject, when AddComponent or Instantiate with arguments is used, so the object will be able to receive its dependencies regardless of game object 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.

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);
```

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);
```

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).

In most cases you don't need to use InitArgs directly; the AddComponent and Instantiate methods Init(args) Lite provides 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 twelve 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 that have been stored for it.

```
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.

Interfaces

IArgs<T...>

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