Dependency Injection

Goal of this package

The goal of this package is to help building a simple yet solid foundation in your project by injecting specific dependencies in places where it's normally tricky to do so. This is **not** a replacement for `SerializeField` during runtime, nor is it a replacement for static classes. Instead it's a way to quickly help keeping your code dependencies in check. Be it Unity or pure C# code. Both can be easily combined with this package.

This package can also manage the life cycle of anything that's `Injectable`. More detailed information about how this works can be found below.

Quick Start

Make something "Injectable"

- Add the `[Injectable]` attribute to any (non-static) class. There's also support for `MonoBehaviours`!
- The `[Injectable]` attribute has three optional parameters to set:
  - `Layer`: Default value is an automatically created based layer. It's possible to create custom `InjectionLayers`. These layers have specific life cycles(defined by you) and will clear all `Injectable` references once a layer is removed and the references are no longer used. This is still a work in progress and is recommended to leave at it's default value for now.
  - `Singleton`: Default value is `true`. Depending on your preferences, you'll probably want to use the Singleton approach most of the time. This way you could create a manager, which is approached as a singleton, keeping track of a bunch of other specific objects in your game/application. Like a "PlayerManager", which knows which players are active/inactive, or simply expose all these players for any other piece of code to use! If a `MonoBehaviour` is injectable as a `Singleton`, it'll automatically be added to `DontDestroyOnLoad` at all times.
  - `ClearAutomatically`: Default value is `false`. If say, a "PlayerManager" is no longer used, this parameter will be checked. If it's value is `true`, the "PlayerManager" object will be destroyed automatically. This can help memory management without you having to keep checking if it's used anywhere, it helps avoiding null reference issues as well! Note that once this object was destroyed, all potentially tracked information will also be lost. So keep this in mind in your approach.

Inject classes/objects when using MonoBehaviours

- Derive your class from CardboardCoreBehaviour instead of MonoBehaviour and implement the required abstracted methods `OnInjected()` and `OnReleased()`.
- Inject your "Injectable" class as a field; `[Inject] private Foo foo;`
- Use `OnInjected()` as an initialization point.
- Use `OnReleased()` as a cleanup point.

Inject classes/objects when using pure C# code (or any other custom setup)

- Use `Injector.Inject(this)` at any initialization moment that fits your needs. The constructor of your class can be a quick start.
- Use `Injector.Release(this)` at any cleanup moment. The **destructor** of your class *could* be a quick start. To make sure there's no lingering injected references lingering around, it's very important that the current class is Released before it's destroyed.
- Inject your "Injectable" class as a field; `[Inject] private Foo foo;`

Good to know

- `Injectable MonoBehaviours` can be added to Unity scenes. This makes it possible for Singletons (the default Injectable behaviour) to have references to other Unity objects such as other MonoBehaviours and ScriptableObjects. Note that once Unity enters play mode, the `Injectable` Singleton `MonoBehaviour` will be added to the `DontDestroyOnLoad` scene.
- Injectable MonoBehaviours can also be left out from scenes. Once any `Injectable` class gets injected anywhere, it'll be automatically created if it's not there yet. This works for both Singletons as non-Singletons. As well as for either Unity-related objects as pure C# classes.
- This package keeps track of all references and cleans up objects that are no longer injected anywhere to clean up memory automatically.
- Inject your dependencies cross-scene!
- It's totally possible to inject Unity objects into pure C# code.

What else?

You'll get a free `Log` tool which is being used to communicate the state of Injected references at this point (a better tool will be added for this soon!). It uses unity's `Debug.Log` under the hood but adds separate colors per logging object and traces down the method the log is being called from.

Use it at your leisure!

```
Log.Write(anyObjectNoNeedToString);

Log.Warn(anyObjectNoNeedToString)

Log.Error(anyObjectNoNeedToString)
```

```
throw new Log.Exception(anyObjectNoNeedToString)
```

Road map

There's support for having multiple `InjectionLayers`. But not a nice way to manage these yet. Support for this tool will be added soon!