DI CONTAINER

**1. Структура контейнера**

Твоя система состоит из 8 файлов:

* **Binding.cs** – хранит данные об одной привязке: какой абстракции соответствует фабрика создания, lifetime, singleton-экземпляр.
* **BindingBuilder.cs** – fluent API для настройки lifetime (AsSingle, AsTransient).
* **Lifetime.cs** – enum с типами времени жизни: Transient (новый объект при каждом Resolve) и Singleton (один объект).
* **IContainer.cs** – интерфейс контейнера.
* **DiContainer.cs** – реализация контейнера (Bind, Resolve, Inject, жизненный цикл, дочерние scope).
* **InjectAttribute.cs** – атрибут для полей/свойств/конструкторов, помечает зависимости.
* **SceneInstaller.cs** – базовый MonoBehaviour, который создаёт контейнер в сцене и автоматически инжектит все GameObject’ы.
* **GameInstaller.cs** – твой конкретный инсталлер, наследуется от SceneInstaller, задаёт биндинги для игры.

## 2. Как работает контейнер

### 2.1 Биндинги (Binding)

Каждая запись говорит: «если кто-то просит **IMoney**, верни MoneyManager».  
Binding хранит:

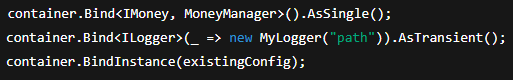
* **Abstraction** – ключ (интерфейс или тип).
* **Factory** – как создать объект.
* **Lifetime** – Singleton или Transient.
* **SingletonInstance** – хранит созданный экземпляр для Singletons.
* **IsExternInstance** – если ты сам передал экземпляр (BindInstance), DI не будет его Dispose-ить.

### 2.2 Привязка (Bind)

В контейнере есть три способа задать зависимость:

* Bind<TAbstraction, TImplementation>() – самый простой, создаст TImplementation при Resolve.
* Bind<TAbstraction>(Func<IContainer,TAbstraction> factory) – если нужен конструктор со своими параметрами.
* BindInstance<T>(T instance) – если уже есть готовый объект.

Все эти методы возвращают **BindingBuilder**, через который выбираешь lifetime:



По умолчанию Transient, т.е. каждый Resolve – новый объект.

### 2.3 Resolve

Resolve<T>() ищет Binding. Если Binding найден:

* **Singleton**: если экземпляр ещё не создан, создаёт и хранит; иначе возвращает старый.
* **Transient**: создаёт новый.
* Инжектит поля/свойства с [Inject].

Если Binding не найден:

* Если это не интерфейс/абстрактный класс – создаёт напрямую, анализируя конструкторы.
* Если есть родительский контейнер – делегирует ему.
* Если ничего не подходит – бросает исключение.

### 2.4 Конструкторы

DIContainer ищет:

* Конструктор с [Inject] (если таких несколько – ошибка).
* Иначе конструктор с наибольшим количеством параметров.
* Параметры конструктора тоже резолвятся через контейнер.
* Если у параметра есть default value, при неудаче подставит его.

### 2.5 Инъекция (Inject)

После создания объекта контейнер проходит по всем полям/свойствам (public/private) с [Inject] и подставляет зависимости.  
Можно вызывать вручную container.Inject(object) или автоматически через SceneInstaller.

Атрибут [Inject(optional:true)] разрешает отсутствие зависимости.

### 2.6 Автоинъекция GameObject’ов

SceneInstaller при Awake() создаёт контейнер, вызывает твой Install(container) и затем:



Это проходит по всем MonoBehaviour в сцене (включая неактивные, если includeInactive=true) и делает Inject для каждого.

### 2.7 Скоупы

CreateChildScope() создаёт контейнер-дочерний с теми же биндингами + своими. Родительские биндинги доступны дочернему, но не наоборот.

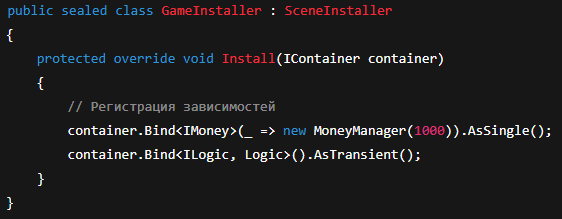
### 2.8 Жизненный цикл и Dispose

Все созданные DI объект, реализующие IDisposable (и не помеченные IsExternInstance), будут автоматически Dispose-нуты при container.Dispose() (SceneInstaller делает это в OnDestroy).

## 3. Как использовать DI в Unity

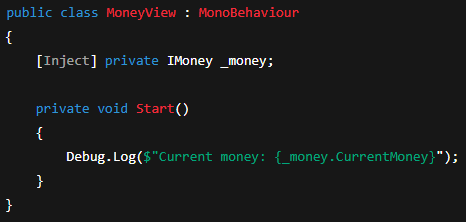
### 3.1 Создай свой Installer

Наследуешься от SceneInstaller:



Добавляешь этот компонент на GameObject в сцене.

### 3.2 Помечай зависимости



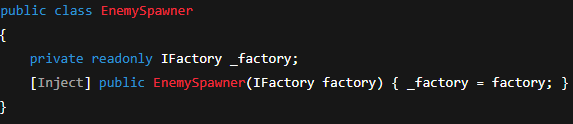
Контейнер при старте сцены сам вызовет Inject и заполнит поле \_money.

Если хочешь, чтобы что-то не упало при отсутствии зависимости:



### 3.3 Инъекция конструктора

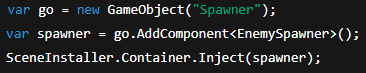
Если у класса есть конструктор:



Контейнер найдёт [Inject] или выберет самый полный конструктор.

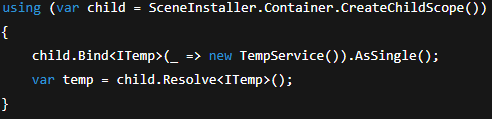
### 3.4 Manual Inject

Если создаёшь MonoBehaviour вручную:



### 3.6 Скоупы

Для временного контейнера:



При выходе из using всё IDisposable освобождается.

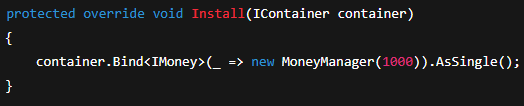
## 4. Практические советы и особенности

* **Cyclic dependency**: контейнер обнаруживает циклы (через \_buildStack) и бросает понятную ошибку.
* **Private поля**: Inject работает и для private, и для public.
* **Lifetime**: Singleton создаётся лениво – при первом Resolve.
* **Dispose**: MonoBehaviour обычно не Dispose-ятся самим контейнером, но ваши сервисы (не компоненты) – да.
* **Debugging**: если видишь NullReference в Start(), значит Inject не успел – проверь, стоит ли твой SceneInstaller до других объектов (он уже стоит DefaultExecutionOrder -10000).

5. Пример связки всего вместе



GameInstaller:



Результат: при старте сцены \_money в MoneyView уже заполнен MoneyManager(1000).