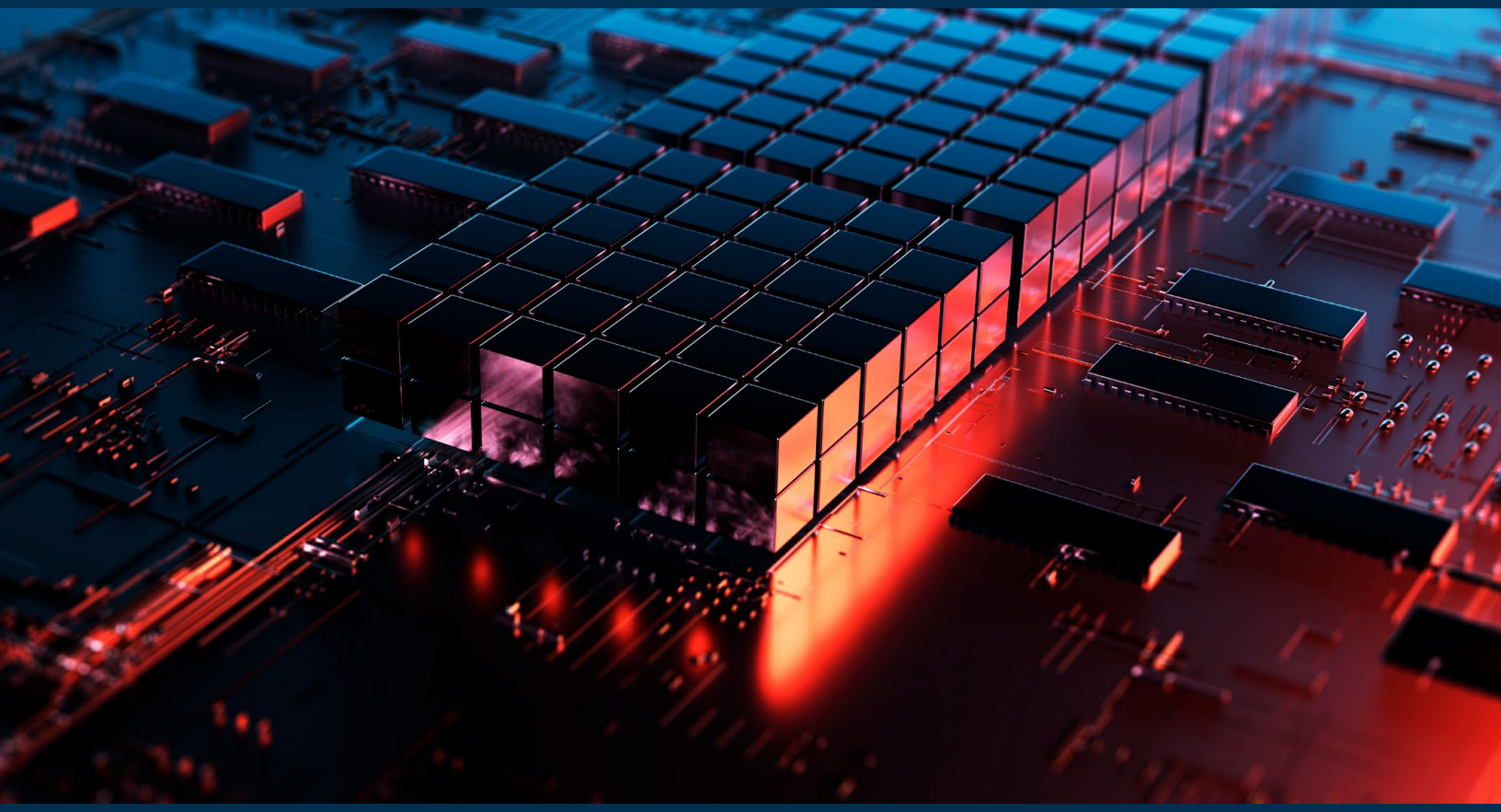
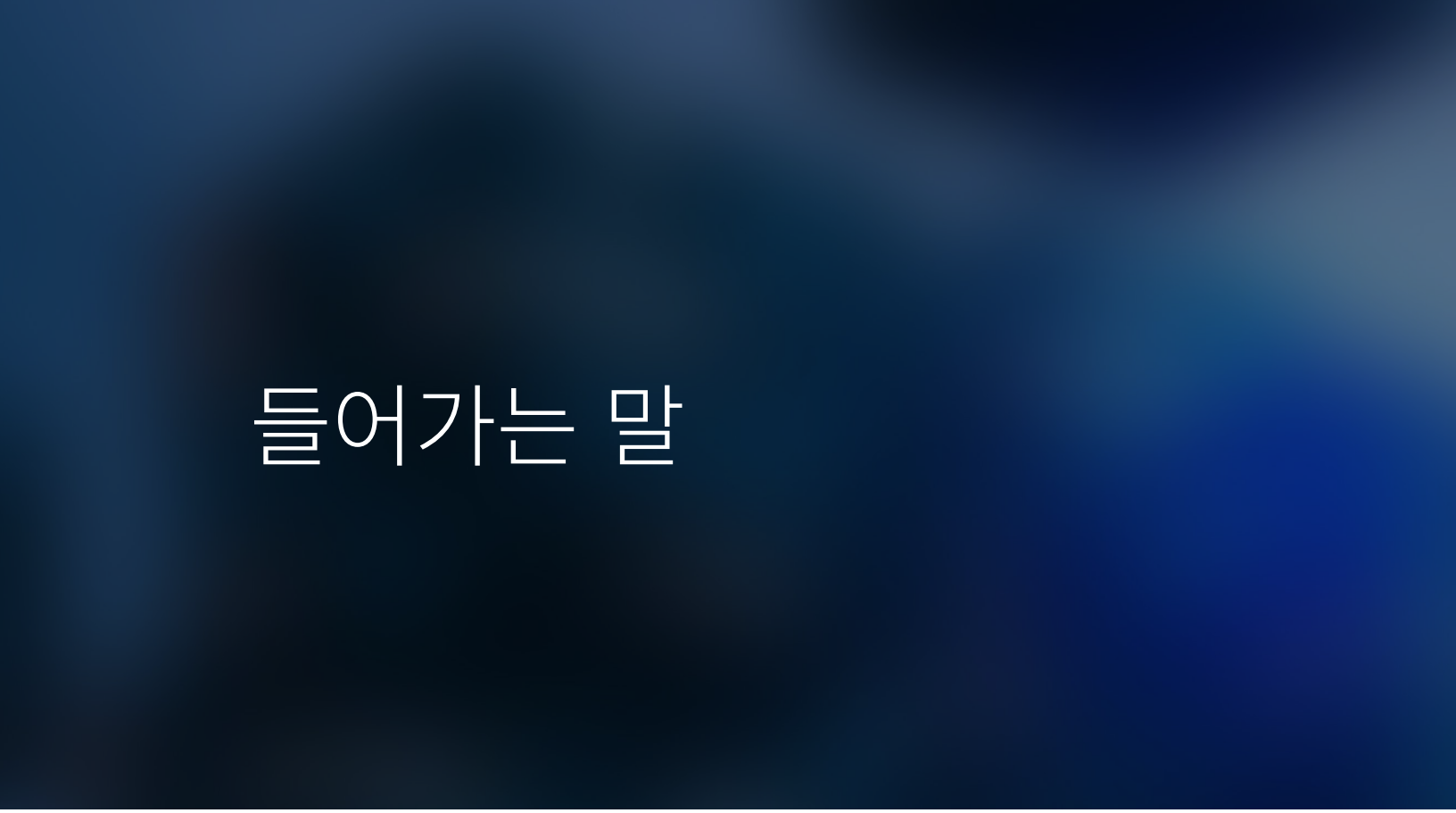
→ 전자책

Unity에서 스크립터블 오브젝트로 모듈식 게임 아키텍처 만들기



Unity®

목차



들어가는 말

스크립터블 오브젝트는 '데이터 컨테이너'라고 표현하는 경우가 많습니다. 그러나 이 표현은 딱 맞아떨어지는 설명은 아닙니다. 스크립터블 오브젝트를 올바르게 적용하면 Unity 워크플로의 속도를 높이고, 메모리 사용량을 줄이고, 코드 아키텍처를 단순화할 수도 있습니다.

이 가이드에는 정식 제작 단계에서 스크립터블 오브젝트를 배포하기 위한 전문 개발자들의 팁과 노하우를 정리했습니다. 특정 디자인 패턴에 스크립터블 오브젝트를 적용하는 방법과 흔히 발생하는 문제를 피하는 방법을 안내하는 예시도 포함되어 있습니다.

스크립터블 오브젝트를 사용하면 데이터를 로직에서 분리하여 코드를 깔끔하게 작성할 수 있습니다. 의도치 않은 부작용을 발생시키지 않으면서 변경 사항을 쉽게 적용할 수 있어 테스트와 모듈식 구성이 모두 용이해집니다.

스크립터블 오브젝트는 에디터에서 인터랙티브 방식으로 사용할 수 있어서 프로그래머와 비프로그래머(아티스트, 디자이너 등) 사이의 협업에 특히 유용합니다. 이 가이드에서는 스크립터블 오브젝트를 사용하는 몇 가지 기법과 방식을 살펴보겠습니다. 기존 워크플로를 보완하고 프로젝트 설정을 간소화하는 데 도움이 되길 바랍니다.

그럼 이제부터 게임 아키텍처의 숨은 영웅, 스크립터블 오브젝트에 대해 알아보겠습니다.

스크립터블 오브젝트란?

스크립터블 오브젝트는 게임 오브젝트 인스턴스에 속하지 않는 Unity 오브젝트입니다. 스크립터블 오브젝트를 사용하면 MonoBehaviour에 비해 더 적은 오버헤드로 자체적인 변수와 메서드를 갖는 커스텀 클래스를 만들 수 있습니다.

스크립터블 오브젝트는 트랜스폼을 갖지 않으며 씬 계층 구조 외부에 있습니다. 그 대신 머티리얼 또는 3D 모델처럼 프로젝트 레벨에 에셋으로 존재합니다.

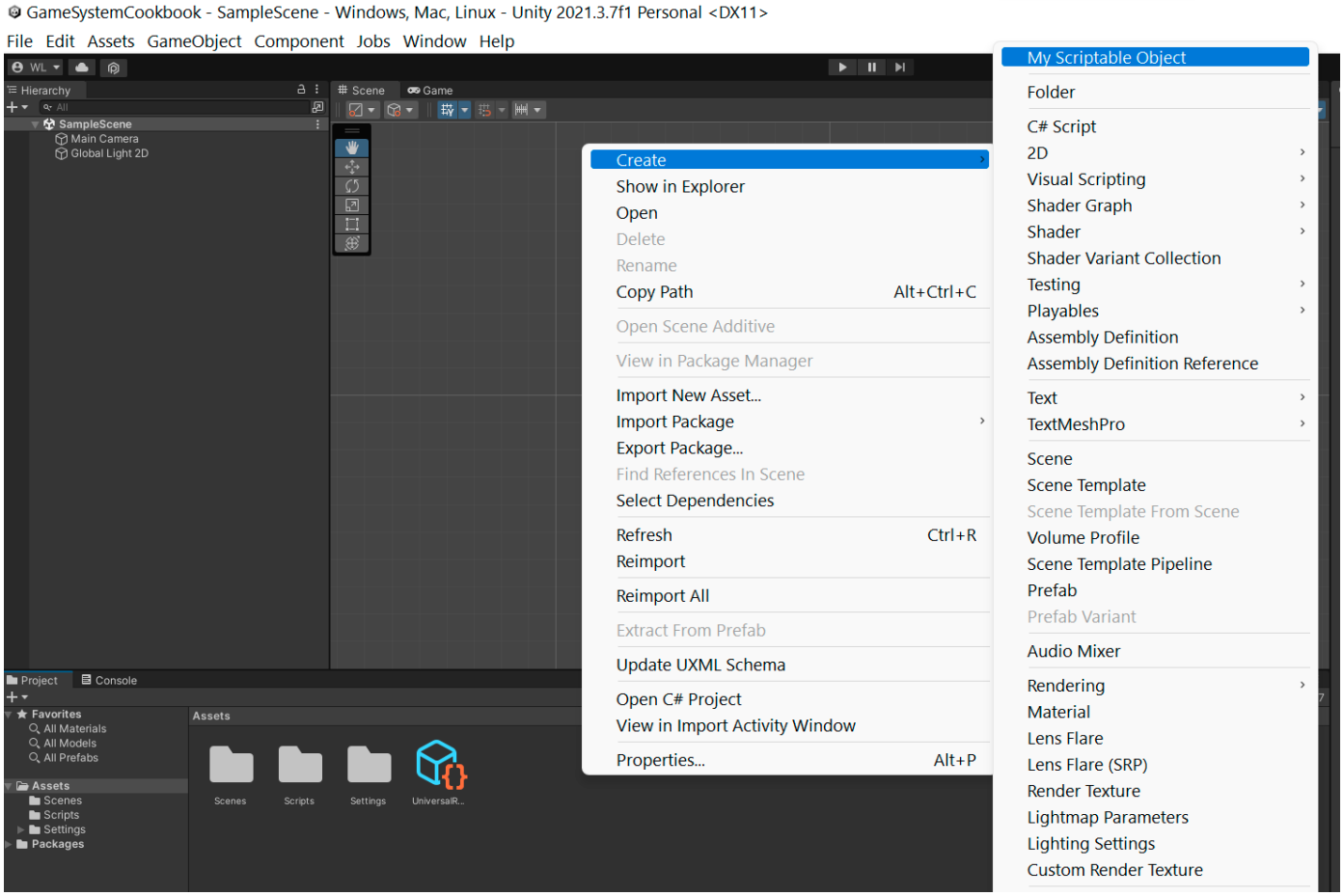
스크립터블 오브젝트는 다음과 같이 선언할 수 있습니다.

```
[CreateAssetMenu(fileName="MyScriptableObject")]
public class MyScriptableObject: ScriptableObject
{
       public int someVariable;
}
```

위 코드에서 볼 수 있듯이, MonoBehaviour에서 파생되는 것이 아니라 `ScriptableObject`에서 상속합니다.

이는 게임 오브젝트에서 직접 사용할 수 없으며, CreateAssetMenu 속성을 통해 메뉴에 제공되는 추가 작업을 사용해야 합니다.

Assets > Create > MyScriptableObject로 이동하거나 프로젝트(Project) 창에서 오른쪽 클릭하면 ScriptableObject 클래스에서 커스텀 에셋을 인스턴스화할 수 있습니다.



스크립터블 오브젝트 생성

그러면 다른 모든 스크립트가 모든 씬에서 필드를 사용하여 이 ScriptableObject 에셋을 참조할 수 있습니다.

```
public class MyMonoBehaviour : MonoBehaviour
{
    public ScriptableObject soInstance;
}
```

스크립터블 오브젝트는 런타임 시 변경될 필요가 없는 항목에 특히 유용합니다.

Unity는 스크립터블 오브젝트를 최우선 오브젝트로 취급하므로 다음과 같은 일을 수행할 수 있습니다.

- 변수에 저장
- 런타임 시 동적으로 생성 또는 파괴
- 인자로 전달
- 메서드에서 반환
- 데이터 구조에 포함
- 직렬화/역직렬화

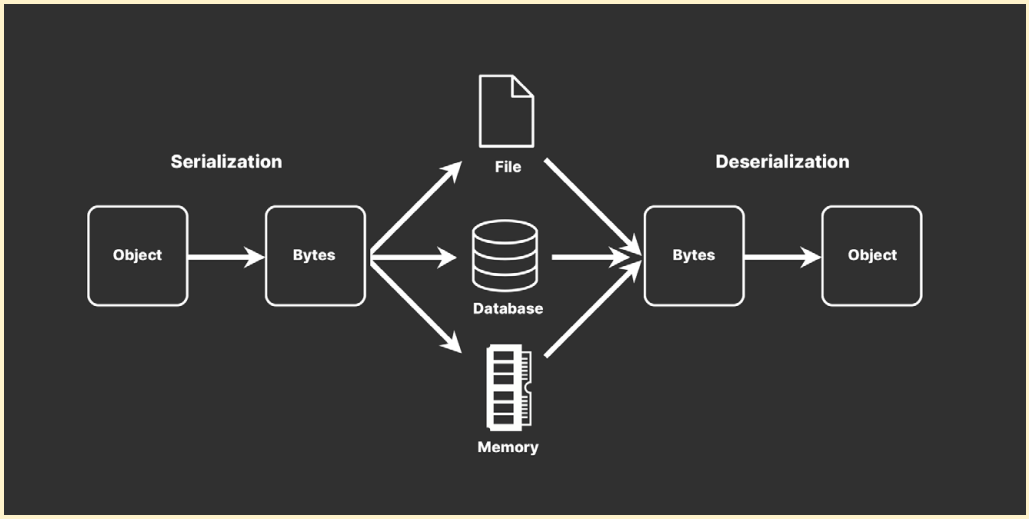
이 중에서 마지막 항목은 스크립터블 오브젝트의 주요 특징 중 하나, 바로 인스펙터에 표시된다는 것을 뜻하기도 합니다. 따라서 스크립터블 오브젝트의 필드는 에디터에서 쉽게 읽고 수정할 수 있습니다.

런타임 시 스크립터블 오브젝트의 값을 변경하면 게임 애플리케이션이 즉시 업데이트되고, 플레이 모드를 종료해도 값이 그대로 유지됩니다. 이는 코드 작성 없이 게임 설정의 밸런스를 조정해야 하는 게임 디자이너들에게 유용합니다. 그뿐만 아니라 스크립터블 오브젝트는 애플리케이션이 실행 중인 동안 변경 사항을 저장할 수 있습니다. 덕분에 MonoBehaviour를 단독으로 사용하는 경우에 비해 몇 가지 이점이 있습니다.

직렬화

직렬화는 데이터 구조나 오브젝트 상태를 쉽게 저장하고 나중에 재구성할 수 있는 포맷으로 변환하는 자동 프로세스입니다. Unity의 직렬화 백엔드는 메모리에 흩어져 있는 데이터를 모아서 순서대로 배치합니다.

이렇게 재구성된 데이터 스트림은 데이터베이스, 파일 또는 메모리에 저장될 수 있습니다. '역직렬화'는 직렬화의 반대입니다.



직렬화는 오브젝트를 저장하거나 메모리, 데이터베이스 또는 파일에 전송하기 위해 해당 오브젝트를 바이트 스트림으로 전환하는 프로세스입니다.

C# 개발에서는 메모리 레이아웃을 간과하기 쉽지만, Unity에는 직렬화를 활용하는 여러 빌트인 기능이 있으므로 이를 숙지해야 합니다.

— **저장 및 로딩:** Unity가 '텍스트 직렬화를 실행'하도록 설정한 상태에서 텍스트 에디터로 **.unity** 씬 파일을 열면 시리얼라이저가 YAML 백엔드로 실행됩니다.

— **인스펙터(Inspector) 창:** 조사 중인 항목의 값을 알아내기 위해 C# API를 이용하는 대신, 오브젝트에 직렬화를 요청한 다음 직렬화된 해당 데이터를 표시합니다.

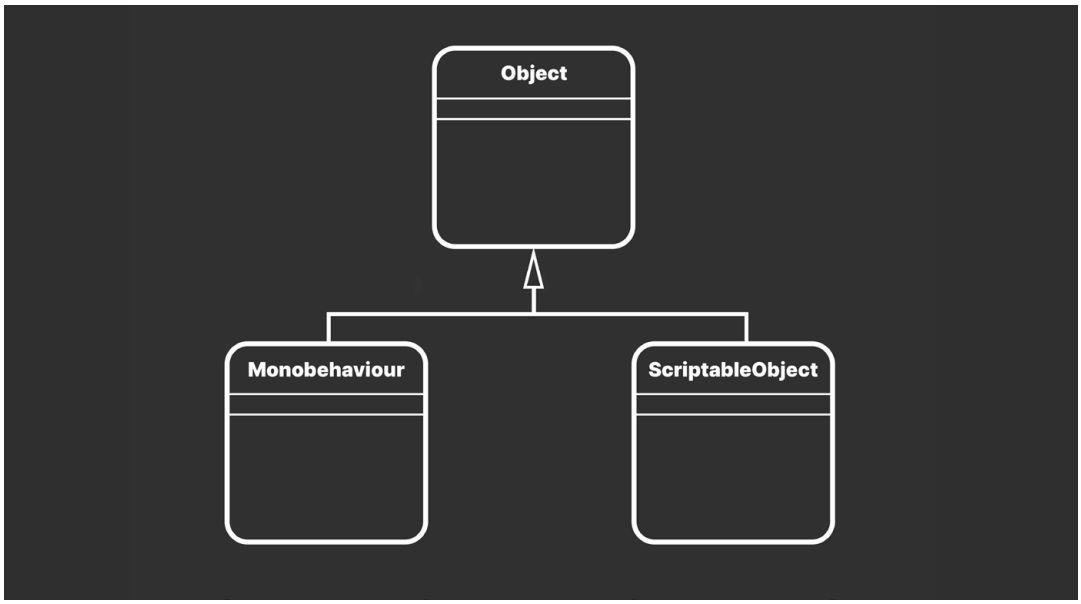
— **프리팹:** 내부적으로, 프리팹은 게임 오브젝트와 그 컴포넌트의 직렬화된 데이터 스트림입니다. 프리팹 인스턴스는 직렬화된 데이터에 적용되어야 하는 일련의 수정 사항입니다.

— **인스턴스화**: 프리팹(또는 씬에 있는 게임 오브젝트)을 인스턴스화하면, 오브젝트가 직렬화되고 새 오브젝트가 생성된 다음 데이터가 새 오브젝트로 역직렬화됩니다.

Unity의 직렬화에 대해 자세히 알아보려면 이 블로그 게시물을 참고하거나 Unity 직렬화 시스템의 작동 방식(영문)을 시청하세요.

ScriptableObject와 MonoBehaviour 비교

겉에서 보기에 ScriptableObject는 단순하며, API에도 몇 개의 메서드밖에 없습니다. 이러한 단순함은 장점이라고 할 수 있습니다. 단순하다는 건 잘못될 가능성이 적다는 의미이기 때문입니다.



Unity 오브젝트 UML

ScriptableObject 클래스는 MonoBehaviour와 마찬가지로 UnityEngine.Object 클래스에서 파생됩니다.

비교

ScriptableObject를 이해하는 가장 좋은 방법은 형제 관계라고 할 수 있는 MonoBehaviour와 비교하는 것입니다. 다음 차트에는 이 두 가지의 유사점과 차이점이 정리되어 있습니다.

MonoBehaviour	ScriptableObject
MonoBehaviour와 ScriptableObject는 둘 다 스크립트입니다. MonoBehaviour 클래스와 ScriptableObject 클래스는 UnityEngine.Object에서 파생됩니다.	
MonoBehaviour는 Unity로부터 콜백을 수신합니다. 메서드를 MonoBehaviour의 이벤트 함수(예: Update())에 따라 명명하여 게임 엔진의 PlayerLoop에 연결합니다. 예: Start, Awake, Update, OnEnable, OnDisable, OnCollisonEnter	**ScriptableObject는 Unity로부터 대부분의 Unity 라이프사이클 콜백(예: Update, Start, FixedUpdate)을 수신하지 않습니다.** ScriptableObject는 런타임 중에 Awake, OnEnable, OnDestroy, OnDisable 같은 소수의 이벤트 함수를 지원합니다. 이에 더해 에디터가 인스펙터에서 OnValidate와 Reset을 호출합니다. ScriptableObject에 대해 그 밖의 메서드를 생성할 수 있지만, 그러한 메서드는 PlayerLoop가 자동으로 호출하지 않습니다.
MonoBehaviour는 런타임 시에 게임 오브젝트에 연결되어 있어야 합니다. 런타임 시에 MonoBehaviour를 생성하려면 AddComponent API를 사용합니다.	**ScriptableObject는 특정 게임 오브젝트에 연결되지 않습니다.** ScriptableObject를 프로젝트 레벨에서 자체 에셋 파일에 저장합니다. 그런 다음 MonoBehaviour 또는 다른 스크립트에서 ScriptableObject 에셋을 참조합니다.
저장할 때는 MonoBehaviour 데이터가 씬과 프리팹에 저장됩니다.	**각 ScriptableObject 인스턴스는 프로젝트 레벨에서 자체 파일에 저장될 수 있습니다.**
에디터에서 플레이 모드를 종료하면 MonoBehaviour 값의 변경 사항이 재설정됩니다.	**에디터에서 플레이 모드를 종료해도 ScriptableObject 값의 변경 사항은 재설정되지 않습니다.** **스탠드얼론 빌드에서 런타임 시에 ScriptableObject 값에 적용된 변경 사항은 저장되지 않습니다.**
MonoBehaviour와 ScriptableObject 둘 다 직렬화 가능하며 인스펙터에서 볼 수 있습니다.	

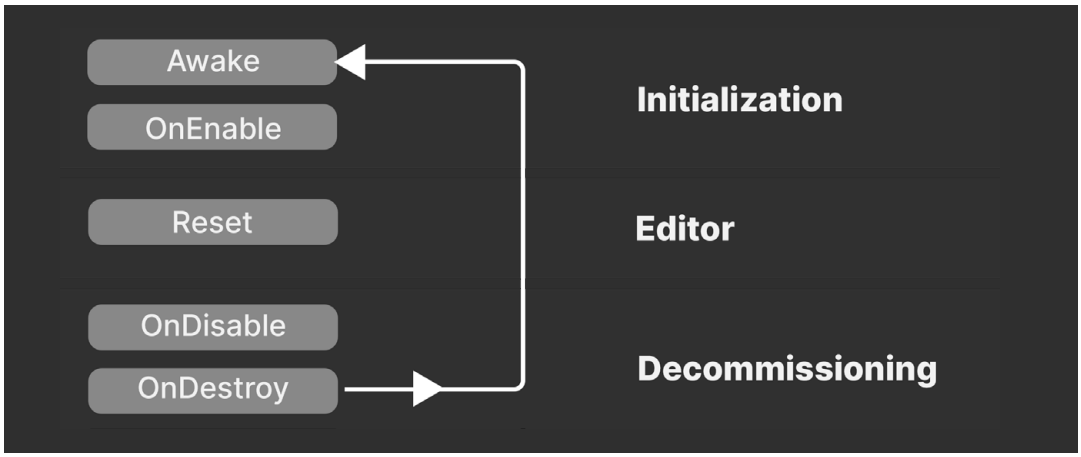
콜백과 메시지

ScriptableObject에는 MonoBehaviour에서 사용할 수 있는 이벤트 함수 하위 집합이 있습니다. ScriptableObject 내에서 자체 메서드를 생성할 수도 있지만, 그러려면 메서드를 직접 호출해야 합니다. 아래 표에는 PlayerLoop에서 자동으로 호출되는 메서드만 정리되어 있습니다.

이벤트 함수 **(런타임)**	**실행 시점**
Awake	MonoBehaviour의 Awake 콜백처럼 ScriptableObject 스크립트가 시작되면 호출됩니다. 게임이 실행되거나 씬이 ScriptableObject 에셋을 참조하여 로드되었을 때도 실행됩니다.
OnEnable	ScriptableObject가 로드되었거나 인스턴스화되었을 때 Awake 콜백 직후에 호출됩니다. ScriptableObject.CreateInstance 중에 또는 성공적인 스크립트 리컴파일 후에 실행됩니다.
OnDisable	ScriptableObject가 범위를 벗어나는 경우 호출됩니다. ScriptableObject는 씬을 ScriptableObject 에셋에 대한 참조 없이 로드할 때 또는 ScriptableObject의 OnDestroy 직전에 범위를 벗어납니다. OnDisable은 스크립트 리컴파일 전에도 실행됩니다. 플레이 모드에 진입한 후에는 OnEnable 직전에 실행됩니다.
OnDestroy	ScriptableObject가 에디터 또는 코드에서 삭제되어 파괴되었을 때 호출됩니다. ScriptableObject를 런타임 시에 생성한 경우, 애플리케이션이 종료되거나 에디터에서 플레이 모드를 종료할 때도 호출됩니다. 참고: 오브젝트의 네이티브 C++ 부분만 파괴됩니다. 자세한 내용은 '생성 및 라이프사이클' 섹션을 참고하세요.
에디터 전용 함수	**실행 시점**
OnValidate	스크립트가 로드되었을 때 또는 인스펙터에서 값이 변경되었을 때 실행됩니다. 데이터를 특정 범위 내로 유지하는 용도로 사용할 수 있습니다.
Reset	인스펙터 컨텍스트 메뉴의 Reset 버튼을 클릭하면 호출됩니다.

ScriptableObject를 파괴하려면 에디터에서 삭제하거나 런타임 시 Destroy/DestroyImmediate를 호출합니다.

다음은 ScriptableObject의 이벤트 함수와 라이프사이클에 대한 간략한 개요입니다. 위에서부터 보면서 MonoBehaviour 이벤트 함수의 실행 순서와 비교해 보세요.



ScriptableObject 이벤트 함수와 실행 순서

파일

MonoBehaviour와 ScriptableObject의 큰 차이점 중 하나는 바로 데이터를 저장하는 방식입니다.

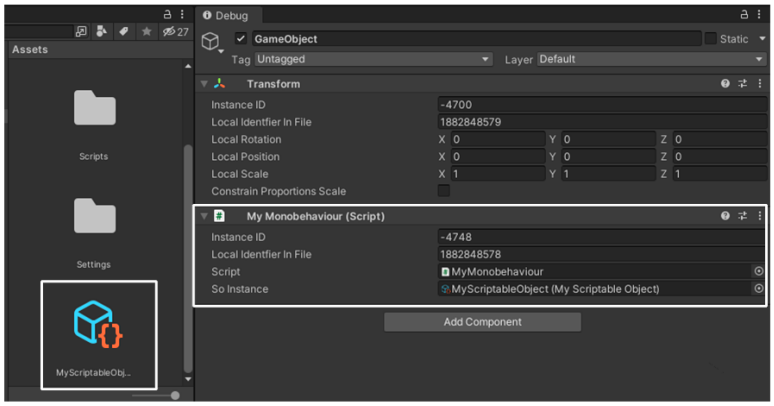
Unity는 씬 또는 프리팹 파일 내에서 MonoBehaviour를 직렬화합니다. 저장된 데이터에 포함되는 내용은 다음과 같습니다.

— MonoBehaviour

— 연결된 게임 오브젝트

— 게임 오브젝트의 트랜스폼

— 연결된 게임 오브젝트의 MonoBehaviour 및 다른 모든 컴포넌트

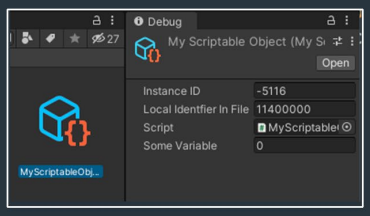


ScriptableObject와 MonoBehaviour 비교

반면에 Unity는 ScriptableObject를 자체 에셋 파일에 저장합니다. 이 파일은 MonoBehaviour에 비해 작고 구조적으로 분리되어 있습니다.

Mode: Force Text를 사용하도록 선택한 경우(**Project Settings > Asset Serialization** 창) 텍스트 에디터에서 ScriptableObject 에셋을 열 수 있습니다. 그러면 다음과 같은 화면이 보입니다.

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!114 &11400000
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 0}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 3fc7a749f692b4f41923450f022d8428, type: 3}
  m_Name: MyScriptableObject
  m_EditorClassIdentifier:
  someVariable: 0

```



텍스트로 직렬화된 ScriptableObject 인스턴스

YAML은 마크업 언어가 아닙니다

Unity는 YAML 사양의 하위 집합을 구현하는 고성능 직렬화 라이브러리를 사용합니다. 이는 XML 및 JSON과 관련 있는 읽기 쉽고 가벼운 언어입니다.

YAML에서는 데이터가 중첩된 요소로 이루어진 계층 구조로 구성됩니다. 각 오브젝트는 클래스 ID, 파일 ID, 오브젝트 유형을 갖습니다. ScriptableObject는 자체적인 유형을 정의하는 대신 오브젝트 유형으로 MonoBehaviour를 사용하는 것을 볼 수 있습니다.

YAML의 오브젝트 헤더

각 오브젝트 아래에는 키-값 페어로 표현되는 직렬화된 프로퍼티가 있습니다.

자세히 알아보려면 Unity의 직렬화 언어, YAML 블로그 게시물을 참고하세요.

생성 및 라이프사이클

ScriptableObject의 라이프사이클은 프로젝트에 있는 다른 모든 에셋(머티리얼, 텍스처 등)의 라이프사이클과 비슷합니다.

앞에 나온 예시처럼 스크립트에 [`CreateAssetMenu`] 속성을 적용하여 에디터에 커스텀 메뉴 작업을 추가합니다. 필요한 경우 기본 fileName 또는 메뉴 항목의 order를 지정할 수 있습니다. 다음 코드는 ScriptableObject 에셋을 생성하는 가장 일반적인 방법입니다.

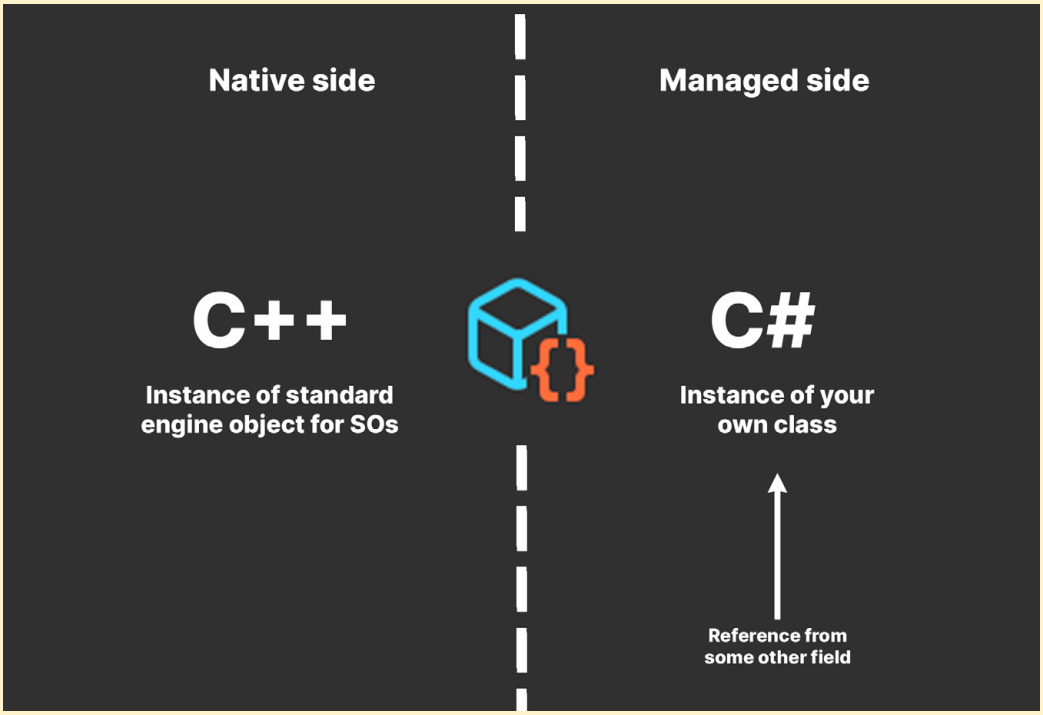
```
[CreateAssetMenu(fileName="MyScriptableObject"]
public class MyScriptableObject: ScriptableObject
{
      public int SomeVar;
}
```

ScriptableObject 인스턴스를 런타임 시에 생성해야 하는 경우, 다음과 같이 정적 `CreateInstance` 메서드를 호출하면 됩니다.

```
ScriptableObject.CreateInstance<MyScriptableObjectClass>();
```

ScriptableObject 파괴

ScriptableObject에는 다른 Unity 오브젝트처럼 네이티브 C++ 부분과 C#으로 관리되는 부분이 있습니다. 네이티브 C++ 부분은 직접 파괴할 수 있으나, 관리되는 부분은 에셋 GC(가비지 컬렉터)가 지울 때까지 남아 있습니다. GC 클린업은 사용자가 씬을 변경하거나 Resources.UnloadUnusedAssets를 호출하면 수행됩니다.



ScriptableObject에는 네이티브 부분과 관리되는 부분이 모두 있습니다.

가비지 컬렉션이 지연되는 상황을 방지하려면 ScriptableObject 에셋에 대한 레퍼런스를 명시적으로 null로 설정합니다.

참고: 이 작업은 `Destroy` 또는 `DestroyImmediate`를 호출하기 전에 수행해야 합니다. 그러지 않으면 오브젝트에 대한 레퍼런스가 실제로는 null이 아님에도 에디터에서 명목상으로 null로 표시될 수 있습니다. GC 클린업은 ScriptableObject에 대한 레퍼런스가 하나도 없는 경우에만 수행됩니다.

ScriptableObject를 생성하고 파괴하는 방법을 숙지했다면, 이제 게임 애플리케이션에서 창의적으로 사용할 수 있는 방법을 알아볼 차례입니다.

데이터 컨테이너

ScriptableObject의 가장 일반적인 용도는 공유되는 정적 데이터의 데이터 컨테이너입니다. 그중에서도 특히 런타임 시 변경되지 않는 게임 설정 데이터의 컨테이너로 널리 사용됩니다.

ScriptableObject의 일반적인 사용 사례는 다음과 같습니다.

- 인벤토리(아이템 유형, 아이콘, 희귀도, 효과 등)
- 적, 플레이어, 항목별 수치(체력, 대미지, 속도, AI 파라미터 등)
- 오디오 컬렉션(발걸음, UI 사운드, 환경 소리 루프를 위한 오디오 클립 그룹 등)
- 설정 파일(난이도 설정, 진행 곡선, 생성 테이블 등)
- 대화 데이터

런타임 시에 이 데이터를 MonoBehaviour에 저장할 수는 있지만, 효율적인 방법은 아닙니다. 앞서 살펴본 비교표에서 볼 수 있듯이, MonoBehaviour가 호스트 역할을 하려면 게임 오브젝트뿐 아니라 기본적으로 트랜스폼까지 필요하므로 더 큰 오버헤드가 발생합니다. 따라서 하나의 값을 저장하기 전에 사용되지 않는 다량의 데이터를 생성해야 합니다.

직접 살펴보려면 빈 MonoBehaviour를 갖는 새 게임 오브젝트를 생성해 보세요. 텍스트 에디터에서 직렬화된 오브젝트를 열면 다음과 같은 화면이 보입니다.

```
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 7467558130563611466}
  - component: {fileID: 2006805663113952451}
  m_Layer: 0
  m_Name: GameObject
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!4 &7467558130563611466
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 338842853706088653}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: -1.1780801, y: -2.6573246, z: -0.01486098}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 0
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
--- !u!114 &2006805663113952451
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 338842853706088653}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 4d1457d233f8d479795a30f859537d5f,
  m_Name:
  m_EditorClassIdentifier:
```

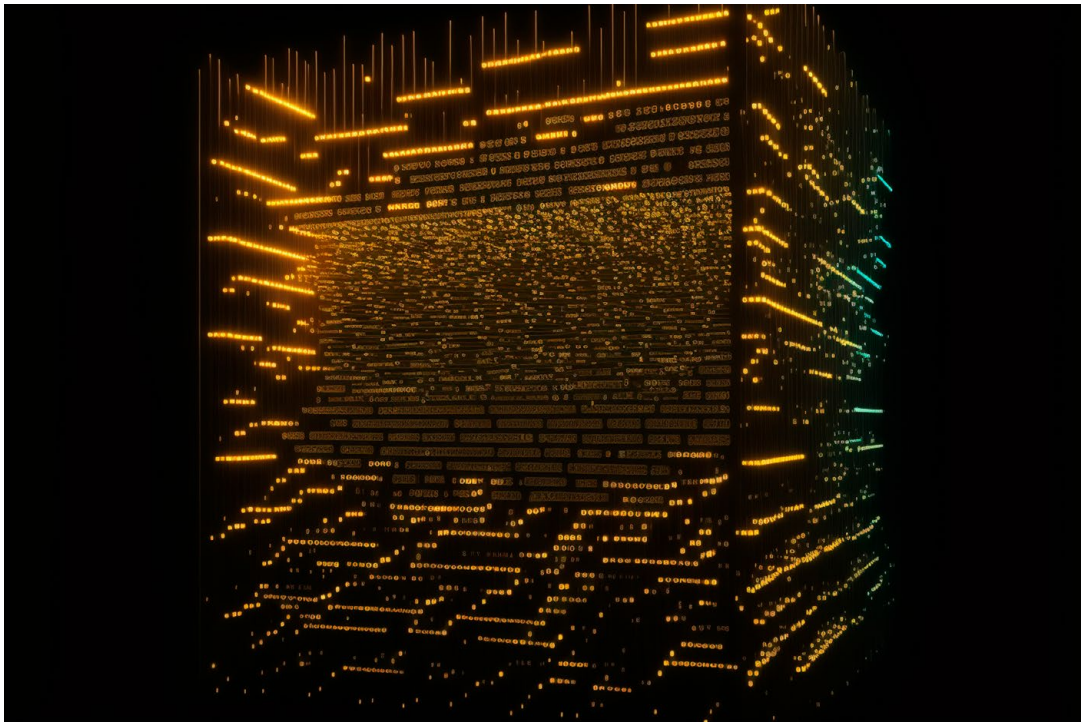
빈 MonoBehaviour를 갖는 새 게임 오브젝트

이를 빈 ScriptableObject와 비교해 보면, 이쪽이 훨씬 간결한 것을 알 수 있습니다.

```
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 0}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: a2d85be56c1ae45d69604547c4544ba5,
  m_Name: PlayerID
  m_EditorClassIdentifier:
```

ScriptableObject는 데이터 저장 시 발생하는 오버헤드를 줄여 줍니다.

ScriptableObject는 메모리 사용량을 줄이고 게임 오브젝트와 트랜스폼을 사용하지 않으므로, 특히 대규모 상용 프로젝트에서 성능을 크게 개선할 수 있습니다. 또한 프로젝트 수준으로 데이터를 저장하며, 이는 여러 씬에서 동일한 데이터에 액세스해야 하는 경우에 특히 유용합니다.



MonoBehaviour에 포함된 추가 데이터는 처음에는 애플리케이션의 성능에 영향을 주지 않을 수 있지만, 게임이 상업적으로 성장하면서 오브젝트가 늘어나면 눈에 띄는 성능 차이를 유발합니다.

스크립터블 오브젝트 데이터와 영구 데이터 비교

스크립터블 오브젝트를 데이터 컨테이너라고 표현할 때, 이 데이터는 보통 런타임 시에 영구적으로 변경할 필요가 없는 정적 또는 공유 설정 데이터를 가리킵니다.

스크립터블 오브젝트 데이터에 대한 변경 사항은 에디터 내에서 유지되지만(머티리얼 또는 프리팹을 수정하는 것처럼), 런타임 시 애플리케이션 빌드에 저장되지는 않습니다. 게임플레이 중에 ScriptableObject 인스턴스에 적용된 변경 사항은 메모리에만 존재하며 세션이 종료되면 사라집니다.

하나의 세션에서 저장되어 다른 세션에 로드되어야 하는 영구 데이터는 보통 다른 파일 포맷(예: JSON, XML, MessagePack, Protocol Buffers)으로 저장됩니다. 자세한 내용은 아래의 '이중 직렬화' 섹션을 참고하세요.

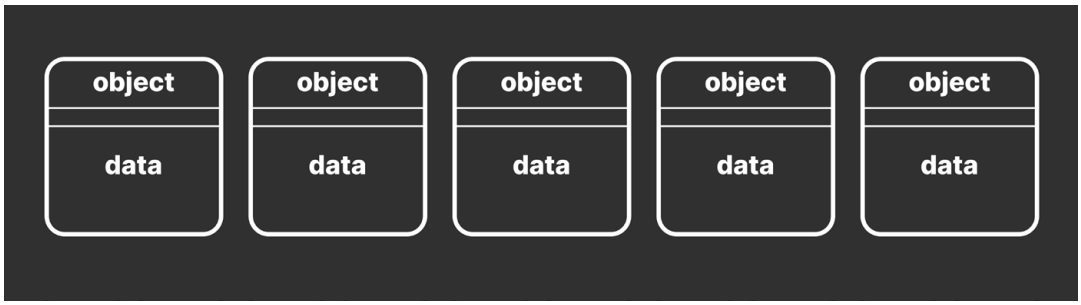
게임 빌드에서 스크립터블 오브젝트 데이터를 런타임 시 변경할 수는 있지만(예: ScriptableObject 변수와 런타임 세트 사용), 이러한 변경 사항은 일시적입니다. 새 게임 세션을 시작하면 스크립터블 오브젝트 데이터가 빌드 시점에 원래 상태로 복원됩니다.

영구 데이터 저장 측면에서 스크립터블 오브젝트 데이터는 '읽기 전용'이라고 생각하세요. 영구 데이터는 '읽기-쓰기'가 모두 가능해야 하며 외부에 저장해야 합니다.

중복 데이터 감소

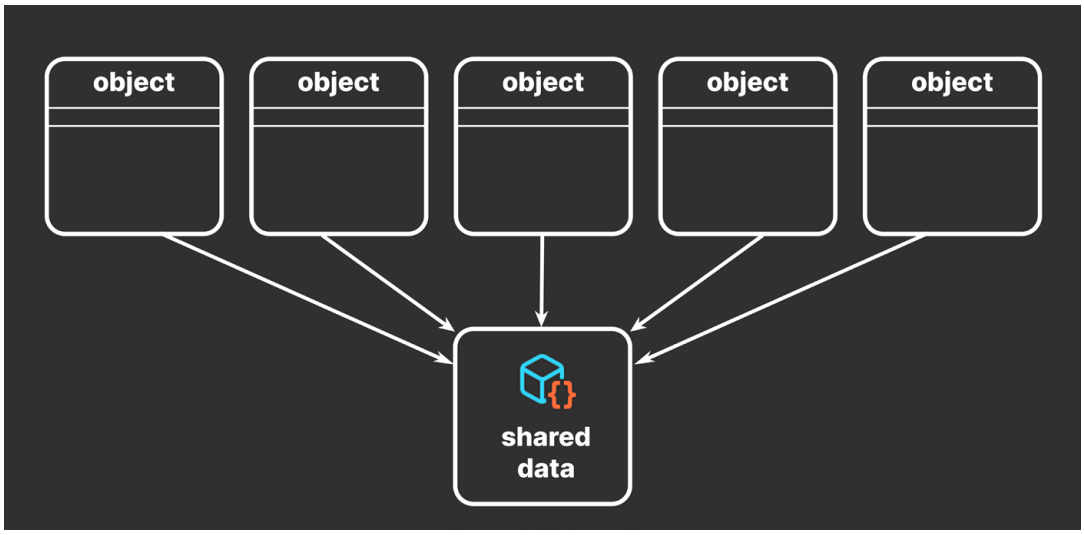
커스텀 MonoBehaviour가 포함된 수천 개의 게임 오브젝트가 있고, 각 MonoBehaviour에는 여러 개의 필드가 있다고 가정해 보겠습니다.

각 컴포넌트가 이러한 값의 사본을 보유하고 있다면 메모리에 엄청나게 많은 중복 데이터가 있는 것입니다. 이러한 상황은 비효율적이며, 인스턴스 간에 데이터가 동일하고 런타임 시 변경되지 않는 경우라면 특히 더 그렇습니다.



중복된 로컬 데이터를 갖는 오브젝트가 많으면 성능 효율이 떨어집니다.

이 정적 데이터를 중복으로 저장하는 대신 스크립터블 오브젝트로 모아 관리할 수 있습니다. 그런 다음 수천 개의 오브젝트 각각이 이 공유 데이터 에셋을 참조하도록 하면 됩니다. 각 오브젝트가 저마다 데이터를 복사하는 대신 데이터에 대한 레퍼런스를 저장하는 것입니다.



스크립터블 오브젝트를 통해 데이터를 공유하는 여러 개의 오브젝트

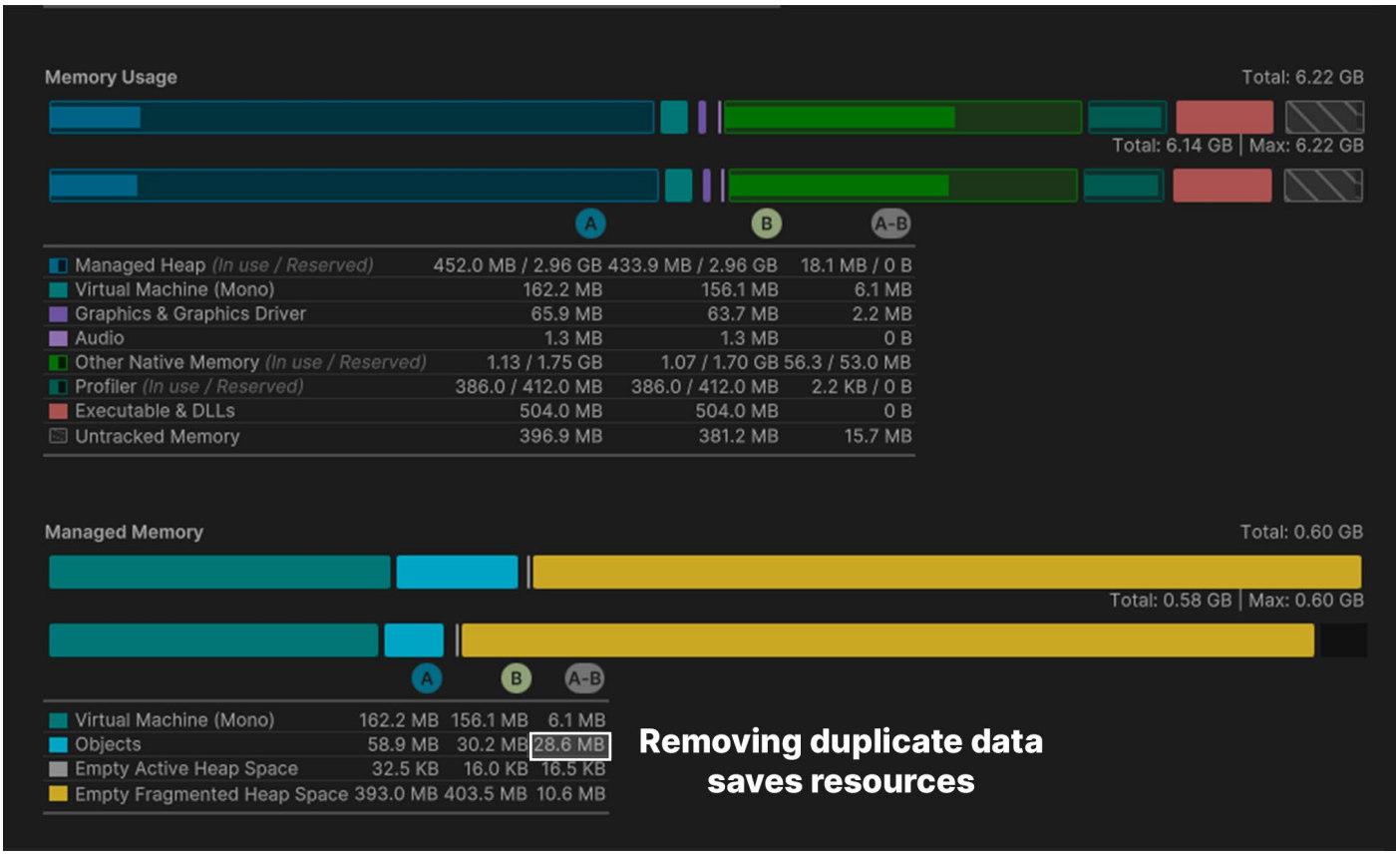
소프트웨어 디자인에서는 이를 플라이웨이트(flyweight) 패턴 최적화라고 합니다. 이렇게 코드를 재구성하면 많은 양의 값이 복사되는 것을 방지하고 메모리 사용량을 줄일 수 있습니다.

디자인 패턴

디자인 패턴을 사용하면 개발자가 더 관리하기 쉽고 유연한 코드를 작성할 수 있습니다. 이는 변경이 잦은 게임 개발 환경에서 특히 유용합니다.

무료 전자책 디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드를 다운로드하고 SOLID 원칙과 디자인 패턴에 대해 자세히 알아보세요.

ScriptableObject에 대한 레퍼런스는 데이터의 전체 사본과 비교하면 훨씬 작습니다. 게임의 규모가 커질수록 데이터를 복제하지 않음으로써 절감하게 되는 메모리는 눈에 띄게 늘어날 것입니다.



Memory Profiler는 중복 데이터(A)와 공유 데이터(B)의 메모리 사용량을 비교합니다.

다량의 공유 데이터를 이 방식으로 저장할 수 있습니다. ScriptableObject를 다음과 같은 용도로 사용해 보세요.

— 에디터 세션 중에 데이터 저장 및 보관

— 데이터를 에셋으로 저장하고 런타임 시에 사용

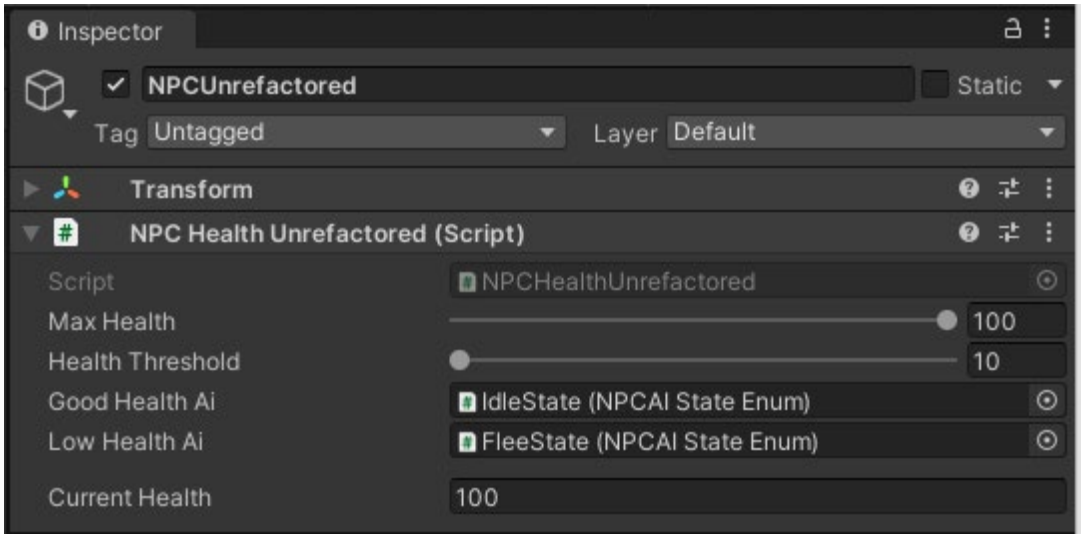
ScriptableObject는 MonoBehaviour와 달리 게임 오브젝트에 연결할 수 없지만, 대신 프로젝트에 에셋으로 저장할 수 있습니다. 이는 MonoBehaviour에서 변경되지 않는 데이터를 사용하는 프리팹이 있는 경우에 특히 유용합니다.

리팩터링 예시

NPC의 체력을 제어하는 MonoBehaviour를 예로 들어 보겠습니다. 클래스는 다음과 같이 정의할 수 있습니다.

```
public class NPCHealthUnrefactored : MonoBehaviour
{
    [Range(10, 100)]
    public int MaxHealth;
    [Range(10, 100)]
    public int HealthThreshold;


    public int CurrentHealth;
}
```

이렇게 해도 작동하긴 하지만, 이 데이터는 런타임 중에 변경되지 않을 것이므로 NPCHealthUnrefactored가 연결된 오브젝트가 많다면 다량의 데이터가 불필요하게 중복될 수 있습니다.



리팩터링 전의 NPCHealth MonoBehaviour

변경할 필요가 없는 데이터는 모두 ScriptableObject로 옮길 수 있습니다.

```
[CreateAssetMenu(fileName="NPCConfig")]
public class NPCConfigSO : ScriptableObject
{
    [Range(10, 100)]
    public int maxHealth;

    [Range(10, 100)]
    public int healthThreshold;


}
```

CreateAssetMenu 속성을 사용하여 메뉴 동작을 설정하세요. 필요한 경우 기본 fileName 또는 메뉴 항목의 order를 지정할 수 있습니다.

본 가이드의 코딩 규칙

본 가이드의 코드 샘플 중 다수는 설명을 위해 쉽게 읽을 수 있도록 단순화되었습니다(예: public 필드).

실제로 정식 제작이 이뤄지면 추가적인 캡슐화와 유연성을 위해 private 필드와 public 프로퍼티를 사용합니다. private 필드를 에디터의 인스펙터에 표시하려면 SerializeField 속성을 적용하세요.

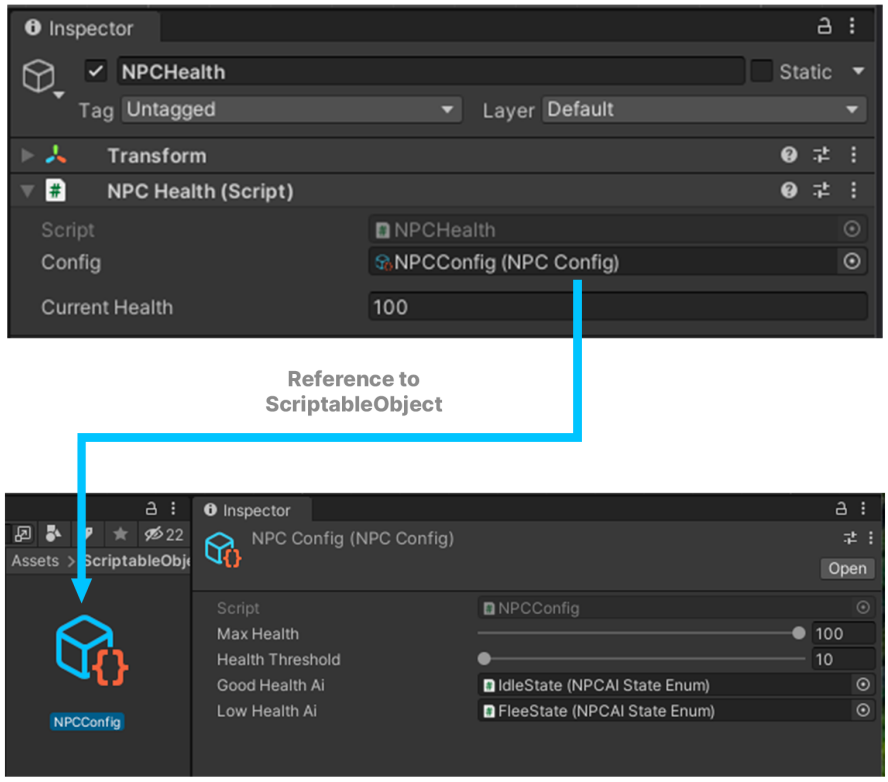
명명 규칙을 사용하면 ScriptableObject 스크립트와 MonoBehaviour를 구분하는 데에도 도움이 됩니다. 한 가지 방법은 클래스 이름 끝에 'Data' 또는 'SO' 접미사를 추가하는 것입니다. 꼭 그래야 하는 것은 아니지만, 이렇게 하면 프로젝트를 일목요연하게 정리하고 모호함을 줄일 수 있습니다.

코드베이스가 커짐에 따라 코드 스타일 가이드를 관리하고 따르는 것이 좋습니다. 자세한 내용은 C# 스타일 가이드로 깔끔하고 스케일링 가능한 게임 코드 작성하기(Unity 6 에디션)를 참고하세요.

리팩터링된 NPCHealth 컴포넌트는 다음과 같이 단순화됩니다.

```
public class NPCHealth: MonoBehaviour
{
    // ScriptableObject에 대한 레퍼런스
    public NPCConfigSO Config;

    public int CurrentHealth;
}
```

MonoBehaviour는 이제 이 새로운 ScriptableObject에 대한 레퍼런스를 포함합니다. 인스펙터에서 보면 리팩터링 후에 다른 모든 것은 그대로인데 데이터가 분할된 것을 볼 수 있습니다.

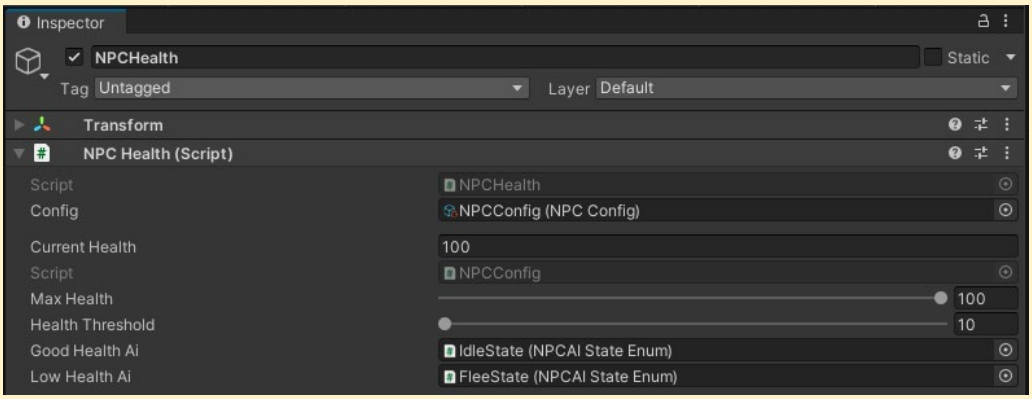


리팩터링은 MonoBehaviour와 ScriptableObject 간에 데이터를 분할합니다.

커스텀 인스펙터

ScriptableObject로 분리한 데이터는 ScriptableObject 에셋과 이를 참조하는 MonoBehaviour, 두 곳에 보관됩니다.

커스텀 에디터를 작성하면 MonoBehaviour를 더 쉽게 찾을 수 있습니다. 다음은 NPCHealth의 예시입니다.



커스텀 인스펙터에 MonoBehaviour의 ScriptableObject 변수가 표시되어 있습니다.

이렇게 하면 NPCConfig의 변수를 MonoBehaviour의 다른 프로퍼티와 함께 살펴볼 수 있습니다. 원본 NPCHealth 프리팹을 선택하면 양쪽 오브젝트의 값을 쉽게 편집할 수 있습니다.

커스텀 에디터에는 다음 동작을 하는 몇 줄의 코드만 있으면 됩니다.

- `Editor`에서 새 클래스를 파생시키고 이를 Editor라는 폴더에 저장합니다. `NPCHealth` 유형에 `CustomEditor` 속성을 적용합니다.
- `NPCConfig ScriptableObject`를 위해 임시 에디터를 예약합니다.
- `OnInspectorGUI` 내에서 `NPCHealth` 컴포넌트를 위한 에디터를 생성합니다.
- 기본 클래스와 새로운 커스텀 인스펙터에서 인스펙터를 드로우합니다.

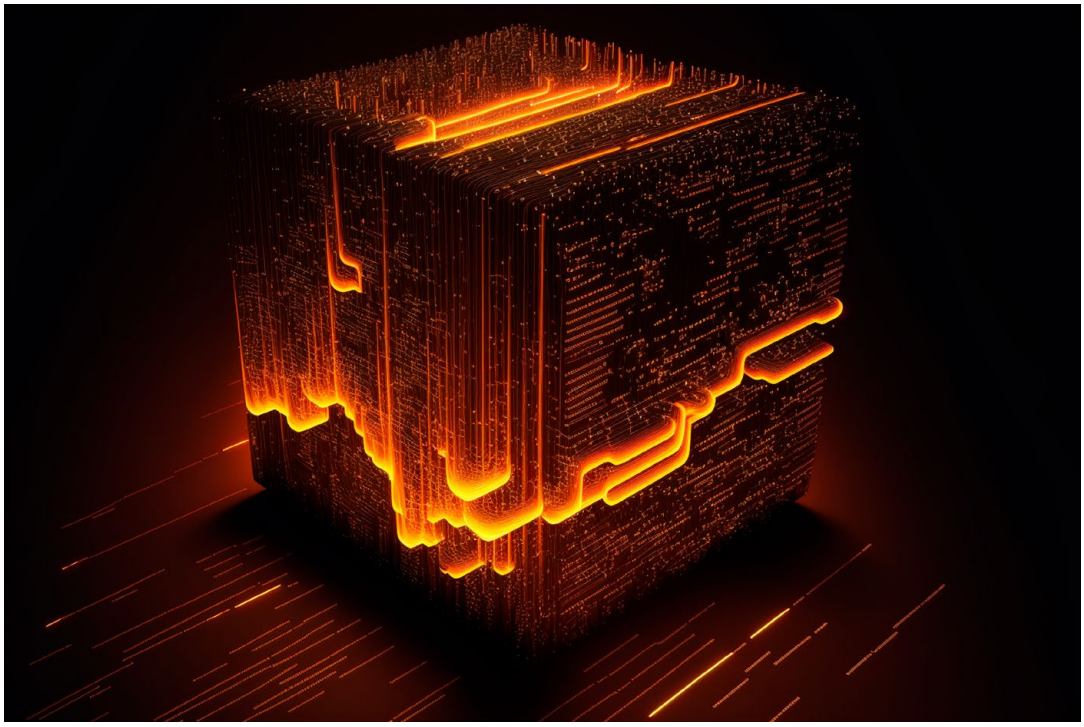
```
using UnityEditor;

[CustomEditor(typeof(NPCHealth))]
public class NPCHealthEditor : Editor
{
    private Editor editorInstance;

    private void OnEnable()
    {
        // 에디터 인스턴스 재설정
        editorInstance = null;
    }

    public override void OnInspectorGUI()
    {
        // 조사 중인 타겟 컴포넌트
        NPCHealth npcHealth = (NPCHealth)target;

        if (editorInstance == null)
            editorInstance = Editor.CreateEditor(npcHealth.config);

        // MonoBehaviour의 변수 표시
        base.OnInspectorGUI();
```

```
        // ScriptableObject 인스펙터 드로우
        editorInstance.DrawDefaultInspector();
    }
}
```

커스텀 프로퍼티 드로어와 에디터 속성을 추가하여 이 예시를 확장할 수 있으며, 이렇게 하면 ScriptableObject를 사용할 때 한층 더 나은 사용자 경험을 제공할 수 있습니다.

아키텍처 측면의 이점

ScriptableObject를 사용하면 공유된 데이터와 공유되지 않은 데이터를 깔끔하게 분리할 수 있습니다. 게임 오브젝트 인스턴스의 고유하고 동적인 데이터는 MonoBehaviour 내부에 유지되고, 공유 데이터는 ScriptableObject에 저장됩니다. ScriptableObject의 아키텍처는 단순한 메모리 절감 외에도 많은 장점을 제공합니다.



코드 아키텍처를 재구성해 얻을 수 있는 몇 가지 장점은 다음과 같습니다.

— **디자이너가 소프트웨어 개발자에게 의존하지 않고 보다 독립적으로 작업할 수 있습니다.** 데이터와 로직을 하나의 MonoBehaviour에 저장하면 개발자와 게임 디자이너가 서로의 작업 영역을 침범하게 됩니다. 이 두 작업자가 동일한 프리팹 또는 씬의 서로 다른 부분을 변경할 경우, 병합 충돌이 발생하여 시간이 낭비됩니다. 공유 데이터를 작은 파일과 에셋으로 분할하면 이러한 문제가 줄어듭니다. 또한 ScriptableObject 아키텍처를 사용하면 디자이너가 프로그래머에게 의존하지 않고도 게임플레이를 빌드할 수 있습니다.

데이터를 공유할 때 각 팀의 워크플로를 명확하게 정의하세요. 효과적으로 소통하고 적정한 경계를 설정하면 이 영역에서의 문제를 방지할 수 있습니다. 이 외에도 오류 검사나 데이터 검증이 필요할 수 있습니다(예: Range 속성이나 OnValidate를 사용하여 잘못된 값 방지).

— **공유된 데이터는 더 빠르게 편집할 수 있고 오류가 발생할 가능성이 더 적습니다.** 공유 데이터의 변경 사항이 한 번에 적용됩니다. 예를 들어 NPC의 설정을 수정해야 한다면 한곳에서 수정한 다음 모든 씬에서 영향을 받는 모든 컴포넌트로 변경 사항을 전파하면 됩니다.

  이렇게 하면 수많은 개별 게임 오브젝트를 하나씩 직접 편집하는 과정에서 발생할 수 있는 오류가 줄어듭니다. 데이터를 ScriptableObject로 오프로드하면 버전 관리도 더욱 원활해지며, 팀원들이 동일한 씬이나 프리팹을 작업할 때 병합 충돌을 방지할 수 있습니다.

— **플레이 모드에서 발생한 게임플레이 수정 사항이 저장됩니다.** 디자이너는 에디터의 플레이 모드를 사용하여 게임플레이와 설정을 테스트해 볼 수 있습니다. 그러나 플레이 모드를 종료하면 Unity는 씬의 임시 사본을 삭제하기 때문에 MonoBehaviour에 적용했던 수정 사항도 사라집니다.

  반면 ScriptableObject는 에셋이기 때문에 플레이 모드 실행 여부와 관계없이 Unity에서 적용한 값의 변경 사항이 저장됩니다. 이는 런타임 중에 값을 조정해야 하는 경우 유용할 수 있습니다.

  그러나 변경 사항을 복원해야 하는 경우에는 문제가 될 수 있습니다. 필요한 경우 언제든지 작업을 복원할 수 있도록 Unity Version Control이나 다른 버전 관리 시스템을 사용하는 것을 잊지 마세요. 자세한 내용은 프로젝트 구성 및 버전 관리 베스트 프랙티스(Unity 6 에디션)를 참고하세요.

— **씬 로딩 시간이 개선됩니다.** 씬 또는 프리팹을 저장하면 Unity가 그 안에 포함된 모든 항목을 직렬화합니다. 여기에는 모든 게임 오브젝트, 게임 오브젝트에 연결된 모든 컴포넌트, 모든 public 필드가 포함됩니다. Unity는 중복 데이터를 확인하지 않고 이 작업을 수행합니다.

  데이터를 ScriptableObject로 옮기면 씬 및 프리팹의 크기를 줄일 수 있으며, 이렇게 하면 로딩 및 저장 시간을 크게 단축할 수 있습니다.

ScriptableObject 변수

하나의 ScriptableObject가 하나의 값만 나타내게 설정하면 공유 데이터 컨테이너를 더욱 세분화할 수 있습니다. 예를 들어 `value`라는 하나의 public 필드를 포함하는 `IntVariable`이라는 ScriptableObject 클래스를 생성할 수 있습니다.

```
using UnityEngine;
[CreateAssetMenu(menuName = "Variables/Int", order = 1)]
public class IntVariableSO : ScriptableObject
{
    public int value;
}
```

그런 다음 MonoBehaviour에서 `IntVariable`을 사용할 수 있습니다. 이렇게 하면 PlayerHealth 클래스를 다음과 같이 구성할 수 있습니다.

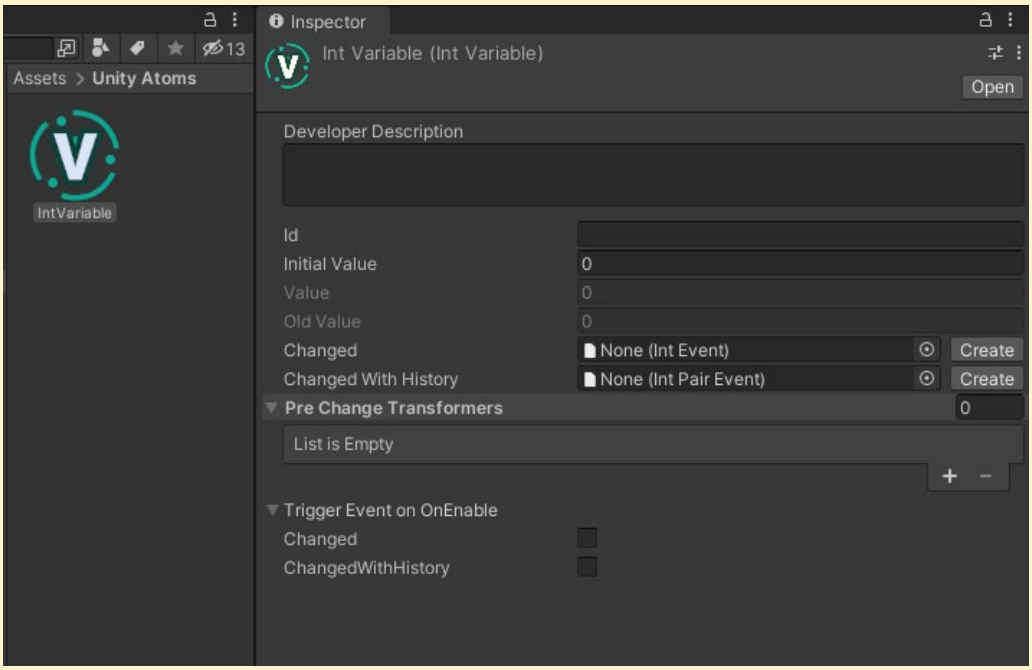
```
public class PlayerHealth : MonoBehaviour
{
    public IntVariableSO health;
}
```

보통 ScriptableObject가 변경되지 않는 값을 저장한다고 생각할 수 있지만, 런타임 중에 이 데이터를 업데이트하고 플레이 모드를 종료할 때 초기 값으로 재설정하는 메서드를 추가하는 방법도 있습니다. 이렇게 하면 정수, 플로트, 부울 등이 포함되어 변수로 작동하는 ScriptableObject를 만들 수 있습니다.

그러면 디자이너가 매번 소프트웨어 개발자에게 요청하지 않고도 게임 로직에 사용할 데이터를 예약할 수 있습니다. 단, 이 방법을 제대로 적용하려면 사전 계획이 필요합니다. 디자이너들과 함께 게임플레이 데이터를 저작(author)하는 작업을 어떻게 분담할지 정하세요.

중요한 것은 협업 방식에 적정한 경계를 설정하는 것입니다. 예를 들어, 프로그래밍 팀이 인벤토리 시스템에 사용하기 위해 ScriptableObject의 초기 설정을 구성한 다음 디자인 팀이 이 설정을 기반으로 각 아이템의 인게임 스탯이나 행동을 채워 넣을 수 있습니다.

여기에 약간의 에디터 스크립팅 작업만 더하면 매끄러운 협업 경험을 구현할 수 있습니다. 또 다른 방식은 인스펙터에서 필드가 ScriptableObject의 공유 값을 사용할지, 상수를 사용할지 전환할 수 있는 기능을 구현하는 것입니다. 이렇게 하면 게임 디자인 팀이 인스턴스별로 ScriptableObject 데이터를 오버라이드할 수 있습니다.



오픈 소스 Unity Atoms 프로젝트의 ScriptableObject 기반 IntVariable 예시

프로젝트에서 이 기능을 구현하는 방법은 유나이트 오스틴에서 진행된 ScriptableObject를 사용하는 게임 아키텍처 프레젠테이션(영문)을 참고하세요. 오픈 소스 Unity Atoms 프로젝트를 다운로드하여 ScriptableObject 변수가 실제로 구현된 방식을 살펴볼 수 있습니다.

이중 직렬화

Unity 내에서 데이터가 직렬화되는 방식을 혼합해서 사용할 수 있습니다. 이렇게 하면 에디터에서 ScriptableObject를 사용하면서 JSON이나 XML 파일과 같은 다른 위치에 데이터를 저장하여 각 포맷의 이점을 십분 활용할 수 있습니다.

JSON, XML과 같은 파일 포맷은 게임의 저장 데이터나 설정과 같은 영구 데이터를 보관하는 데 적합합니다. 에디터에서 사용하기는 불편할 수 있지만, Unity 외부에서 텍스트 에디터로 쉽게 수정할 수 있습니다.

반면에 ScriptableObject는 에디터에서 쉽게 사용할 수 있고 간단한 드래그 앤 드롭 조작으로 교체할 수 있습니다. 단, Unity 외부에서 수정하거나 플레이어 커뮤니티에 공유하기는 불편합니다.

직렬화된 포맷을 혼합해서 사용하면 게임에 레벨 편집, 모딩과 같은 새로운 가능성을 더할 수 있습니다. 스크립트를 사용하면 빌드 시점에 다른 파일을 일반 텍스트보다 로딩이 빠른 ScriptableObject로 전환할 수 있습니다.

일부 민감한 데이터(예: 가상 통화, 계정 정보)는 서버에 안전하게 저장해야 하지만, 게임 데이터의 일부를 커뮤니티에 공개하면 사용자가 자유롭게 활용할 수 있는 샌드박스 형식의 레벨을 제공하여 게임플레이의 수준을 높일 수 있습니다.

게임 레벨 레이아웃을 정의하는 ScriptableObject를 떠올려 보세요. 여기에는 단순히 프리팹 배치, 초기 설정 등을 정의하는 일련의 트랜스폼이 포함되며, 게임 스크립트에서 이 데이터를 사용하여 각 레벨을 구성합니다.

게임의 벽과 시작 위치가 ScriptableObject에 저장되어 있다고 가정해 보겠습니다.

```
[CreateAssetMenu(fileName = "LevelLayout")]
public class LevelLayout : ScriptableObject
{
    public Vector3[] wallPositions = new Vector3[2];
    public Vector3[] playerPositions = new Vector3[2];
    public Vector3[] goalPositions = new Vector3[2];
    public Vector3 ballPosition;
}
```

이는 레벨을 설정하는 방식을 정의합니다. 레벨 관리 스크립트는 LevelLayout 오브젝트에서 데이터를 읽은 다음 프리팹을 올바른 위치에 인스턴스화할 수 있습니다.

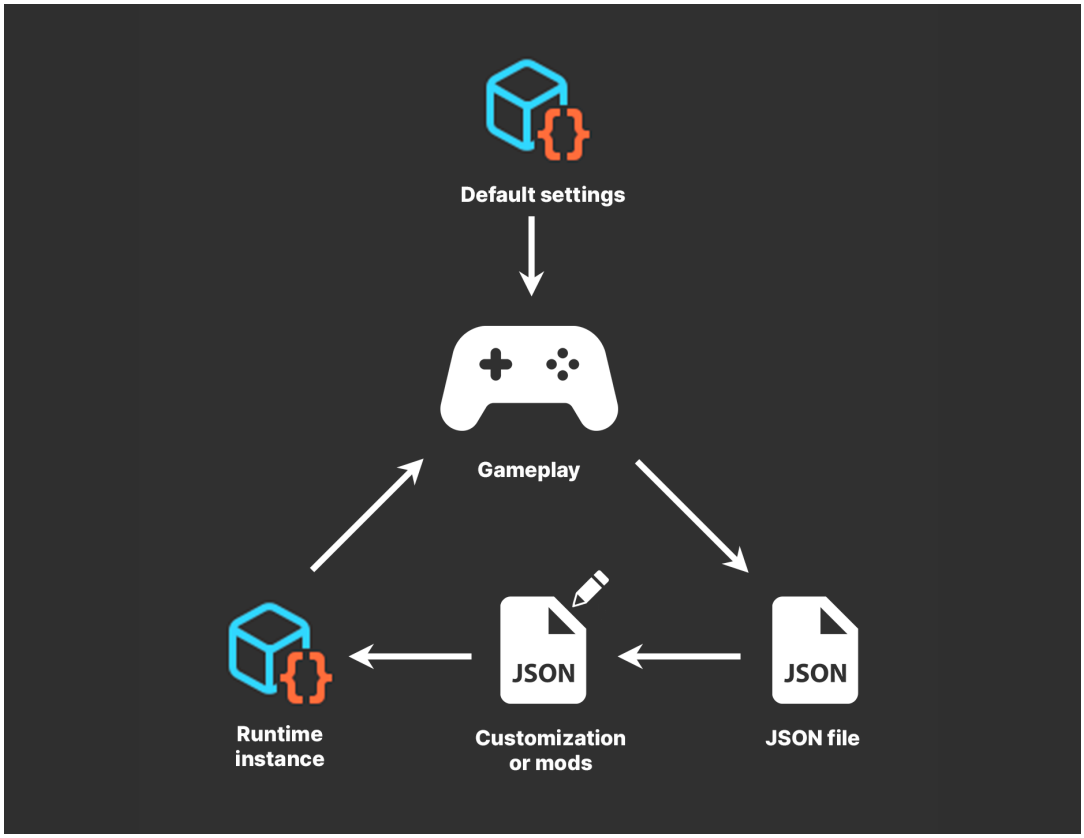
커스텀 스크립트는 JsonUtility를 사용하여 이 데이터를 디스크로 익스포트할 수 있습니다. 그러면 에디터 외부에 사용자들이 외부 툴을 사용하여 수정할 수 있는 텍스트 파일이 생성됩니다.

커스텀 모딩된 레벨을 로드하기 위해, `ScriptableObject.CreateInstance`는 런타임 시 ScriptableObject를 생성한 다음 JSON 파일에서 텍스트를 읽고 ScriptableObject를 채울 수 있습니다. 다음 `LoadLevelFromJson` 메서드 예시를 살펴보세요.

```
using System.IO;

public class LevelManager : MonoBehaviour
{
    public ScriptableObject levelLayout;

    public void LoadLevelFromJson(string jsonFile)
    {
        if (levelLayout == null)
        {
            levelLayout = ScriptableObject.CreateInstance<LevelLayout>();
        }

        var importedFile = File.ReadAllText(jsonFile);
        JsonUtility.FromJsonOverwrite(importedFile, levelLayout);

    }
}
```

커스텀 데이터가 ScriptableObject의 콘텐츠를 대체하고, 따라서 외부에서 모딩된 레벨을 게임에서 다른 일반적인 레벨처럼 사용할 수 있게 됩니다. 애플리케이션은 차이를 감지하지 못합니다.

샘플 프로젝트에서 이 과정을 직접 살펴보세요. 수정된 JSON 파일을 로드하면 커스터마이즈된 레벨이 ScriptableObject의 기본 레벨 데이터를 오버라이드합니다.



직렬화된 포맷을 혼합해서 사용하면 유연성이 향상됩니다.

참고: JsonUtility를 사용하여 JSON을 ScriptableObject로 역직렬화할 때는 FromJsonOverwrite 메서드를 사용해야 합니다.

새 오브젝트를 생성하고 해당 오브젝트에 JSON 데이터를 로드하는 대신 JsonUtility는 JSON 데이터를 기존 오브젝트에 로드합니다. 그러면 클래스나 오브젝트에 저장된 값이 할당 없이 업데이트됩니다.

데이터 보호

위에서 살펴본 간단한 모딩 예시는 ScriptableObject의 다양한 응용 사례 중 하나일 뿐입니다. 그러나 수정할 수 있도록 데이터를 공개할 때는 플레이어들이 애플리케이션의 나머지 부분을 변조하지 못하도록 주의를 기울여야 합니다.

모딩되면 안 되는 데이터를 보호하는 일반적인 방법은 다음과 같습니다.

- **암호화:** 암호화를 사용하여 데이터 파일이 쉽게 읽히거나 수정되지 않도록 보호합니다. 이렇게 하면 사용자들이 중요한 데이터를 변경하기가 한층 어려워집니다.
- **디지털 서명:** 지문 알고리즘을 사용하여 데이터 파일의 변조 여부를 확인할 수 있습니다.
- **서버 측 검증**: 게임에서 서버에 저장된 데이터를 사용하는 경우, 데이터가 게임에서 사용되기 전에 서버 측에서 데이터를 검사하고 조작된 것으로 보이는 데이터를 거부하는 방식입니다.

한 가지 방법으로 모든 문제를 방지할 수는 없으며, 플레이어들로 인해 게임에 버그나 취약점이 생기지 않도록 다양한 기법을 함께 사용하는 것이 좋습니다.

확장 가능한 열거형 패턴

게임을 개발하다 보면 반복적이거나 비슷한 문제를 해결해야 하는 경우가 많습니다. 이때 디자인 패턴을 사용하면 이미 비슷한 문제를 겪은 소프트웨어 엔지니어들의 집단 지성을 활용할 수 있습니다.

디자인 패턴은 확장 가능한 대규모 애플리케이션을 빌드하도록 지원하는 범용 솔루션으로, 코드의 가독성을 높여 주고 코드베이스를 깔끔하게 만들어 줍니다. 디자인 패턴은 리팩터링과 테스트 소요 시간을 줄이는 데에도 도움이 됩니다.

다음과 같은 일반적인 과제를 해결하기 위한 템플릿이라고 생각하세요.

- 효율적으로 대량의 데이터 저장
- 서로 다른 게임 시스템의 오브젝트 간 통신 지원
- 런타임 중에 실시간으로 동작 교체

스크립터블 오브젝트를 사용하면 이 몇 가지 패턴을 구현할 수 있습니다. 앞에서 스크립터블 오브젝트가 데이터 컨테이너로 작동하는 방식을 살펴보았는데, 사실 스크립터블 오브젝트는 단순히 값이나 설정을 저장하는 것보다 더 많은 기능을 할 수 있습니다. 이어지는 섹션에서는 스크립터블 오브젝트로 데이터 저장 외에 다양한 작업을 수행하는 방법을 살펴봅니다.

열거형 카테고리

꼭 데이터 컨테이너로 쓰지 않더라도 스크립터블 오브젝트는 유용합니다. 실제로 빈 스크립터블 오브젝트를 생성해 보면 그 유용성을 확인할 수 있는데, 다른 스크립터블 오브젝트와 비교하는 용도만으로 쓰더라도 그렇습니다.

다음과 같이 게임 애플리케이션의 빈 GameItemSO 스크립터블 오브젝트에서 여러 개의 에셋을 만들었다고 가정해 보겠습니다.

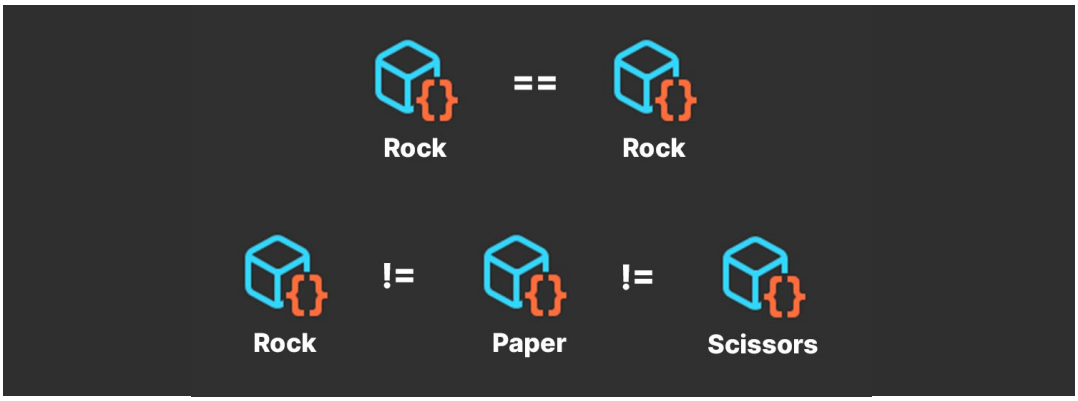
```
Using UnityEngine;

[CreateAssetMenu(fileName="GameItem")]
public class GameItemSO : ScriptableObject
{

}
```

빈 ScriptableObject는 열거형으로 작동합니다.

따라서 프로젝트 내에서 에셋을 원하는 만큼 생성할 수 있습니다. 데이터를 포함하지 않는 스크립터블 오브젝트는 열거형처럼 그 자체로 카테고리나 항목 유형을 나타낼 수 있습니다.

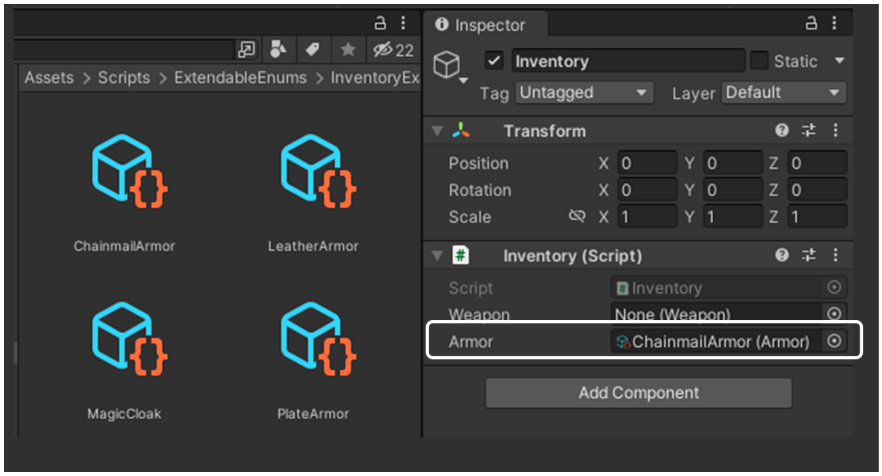
두 개의 변수가 하나의 스크립터블 오브젝트를 참조한다면 두 변수의 항목 유형이 동일한 것이고, 그러지 않는다면 항목 유형이 서로 다른 것입니다.

특수 대미지 효과(냉기, 열, 전기, 마법 등)를 정의하는 스크립터블 오브젝트가 있을 수도 있고, 제로섬 게임의 가위바위보 할당을 정의하는 스크립터블 오브젝트가 있을 수도 있습니다.



스크립터블 오브젝트 비교

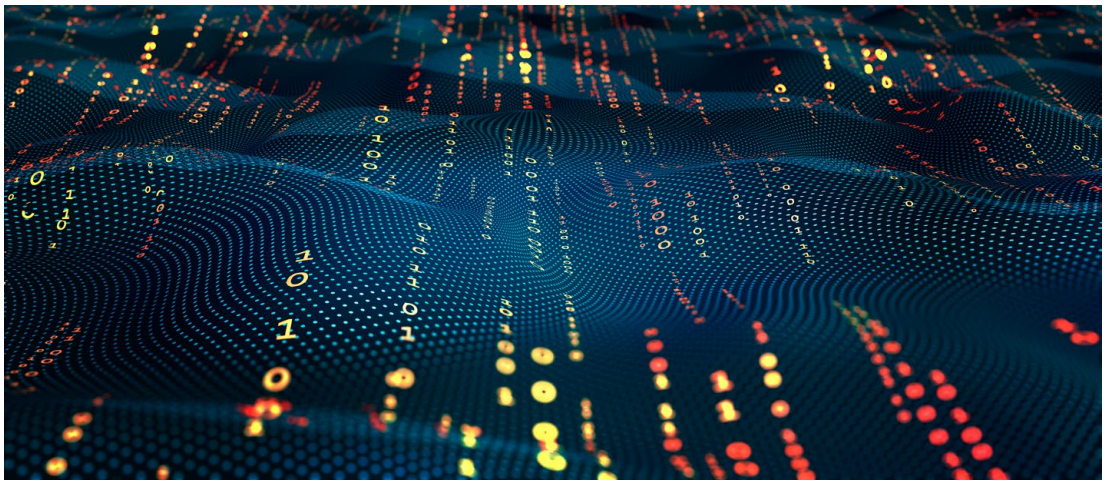
애플리케이션에 게임플레이 아이템을 보관할 인벤토리 시스템이 필요하다면 스크립터블 오브젝트로 아이템 유형이나 무기 슬롯을 나타낼 수도 있습니다. 이 경우 인스펙터의 필드는 아이템 장착을 위한 드래그 앤 드롭 인터페이스로 작동합니다.



Inventory slots categorized by ScriptableObject

드래그 앤 드롭 ScriptableObject 기반 카테고리

이 UI는 아티스트에게도 친숙하므로 디자이너들이 개발자의 도움 없이 게임플레이 데이터를 수정하고 확장할 수 있으며, 디자인 팀에 게임플레이 데이터를 관리할 수단과 책임을 부여함으로써 모든 팀원이 가장 잘하는 일에 집중하게 됩니다.



동작 확장

스크립터블 오브젝트에 더 많은 데이터를 추가하여 확장하려는 경우, 스크립터블 오브젝트를 열거형으로 사용하면 한층 더 유용할 수 있습니다. 스크립터블 오브젝트는 일반적인 열거형과 달리 추가 필드와 메서드를 가질 수 있습니다.

다음은 수정된 가위바위보 `GameItem`입니다. 여기에서도 `ScriptableObject` 에셋이 열거형 카테고리를 정의하지만, 더 이상 비어 있지 않습니다.

```
public class GameItem : ScriptableObject
{
    public GameItem weakness;
    public bool IsWinner(GameItem other)
    {
        return other.weakness == this;
    }
}
```

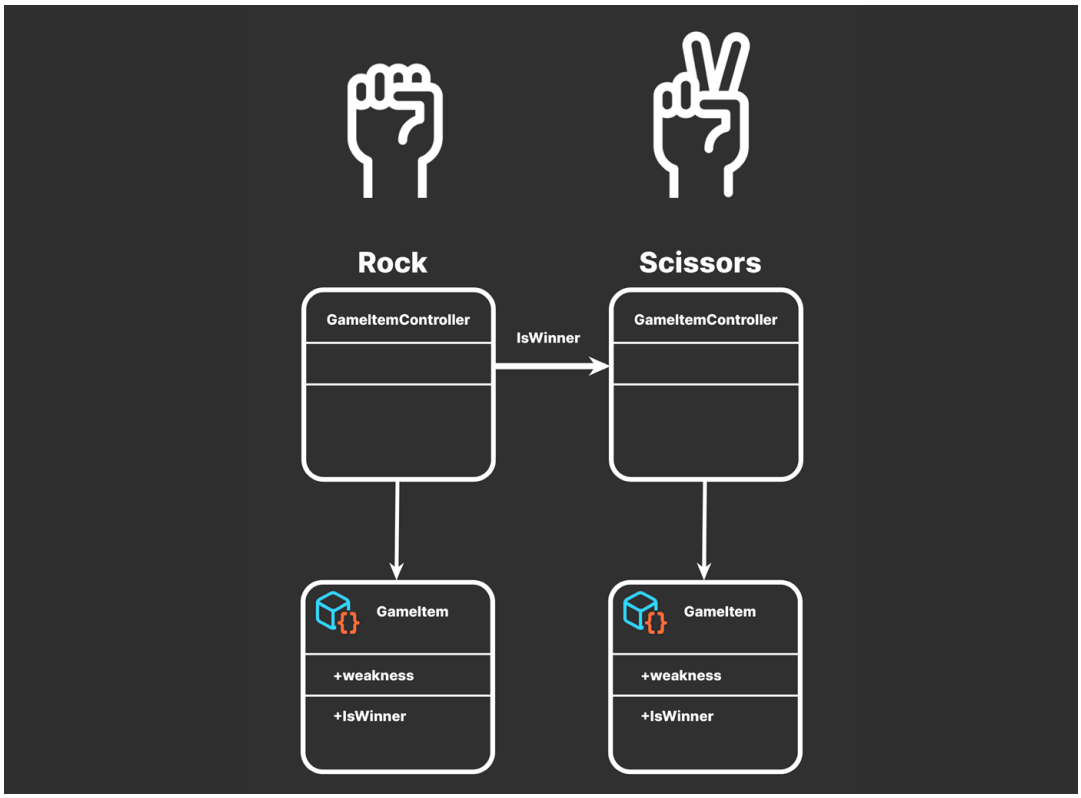
이제 ScriptableObject에는 게임에서 상호 작용의 결과로 어느 항목이 승리하는지를 정의하는 weakness 필드가 있습니다. 데이터를 저장하는 것 외에도 각 ScriptableObject는 `IsWinner`에 간단한 비교 로직을 포함하게 됩니다.

이제 각 게임플레이 아이템에 특정 ScriptableObject 에셋을 참조하는 MonoBehaviour가 필요합니다. 다음 예시는 컨트롤러 스크립트로 작동합니다.

```
public class GameItemController : MonoBehaviour
{
    // 가위, 바위, 보
    public GameItem gameItem;

    private void OnTriggerEnter(Collider other)
    {
        GameItemController otherController = other.GetComponent<GameItemC
ontroller>();
        GameItem otherGameItem = otherController.gameItem;

        if (gameItem.IsWinner(otherGameItem))
        {
            Debug.Log(gameItem.name + " beats " + otherGameItem.name);
        }
    }
}
```

이 스크립트는 ScriptableObject를 필드로서 참조합니다. `OnTriggerEnter`에서 `IsWinner` 메서드를 보면 gameItem이 다른 아이템과 접촉했을 때 어느 쪽이 승리하는지 알아볼 수 있습니다. 하지만 이로 인해 가위바위보처럼 승패가 엇갈리는 대결 구도가 발생할 가능성이 생깁니다.

스크립터블 오브젝트는 일반적인 열거형과 달리 쉽게 확장할 수 있습니다. 별도의 룩업 테이블을 사용하거나 새로운 데이터 배열과의 상관 관계를 만들 필요 없이 로직을 처리할 필드나 메서드만 추가하면 됩니다.



비교 로직이 포함된 스크립터블 오브젝트
출처: Flaticon

기존 열거형의 관리 방식과 비교하여 생각해 보세요. 명시적인 번호가 부여되지 않은 긴 열거형 값 목록에서 열거형 항목 하나를 삽입하거나 삭제하면 순서가 바뀔 수 있습니다. 이렇게 순서가 변경되면서 미세한 버그가 생기거나 의도치 않은 동작이 발생할 수 있습니다.

스크립터블 오브젝트 기반 열거형에는 이런 문제가 없습니다. 새로운 항목을 프로젝트에 추가하거나 기존 항목을 삭제해도 문제없이 작동합니다.

RPG에서 아이템을 장착 가능하게 만들려면 스크립터블 오브젝트에 부울 필드를 새로 추가하면 됩니다. 특정 캐릭터가 특정 아이템을 사용할 수 없도록 하거나, 특정 아이템에 마법 효과나 특수 능력을 부여하려는 경우에도 스크립터블 오브젝트 기반 열거형으로 간단하게 처리할 수 있습니다.

이런 식으로 게임 디자인을 구현하는 과정에서 게임플레이 데이터를 발전시켜 나갈 수 있습니다. 초기에 필드를 설정하는 방법을 정해 두면 이후에 디자이너가 스스로 세부 사항을 채워 넣을 수 있습니다.

패턴:
델리게이트 오브젝트

Unity의 스크립터블 오브젝트는 데이터를 저장할 뿐 아니라 그 안에 메서드를 구성할 수도 있습니다. 즉, 해야 하는 일(로직)과 사용할 재료(데이터)를 모두 담을 수 있는 것입니다.

델리게이트와 이벤트 비교

델리게이트와 이벤트는 C#에서 긴밀하게 연관되어 있는 개념이지만, 그 용도는 서로 다릅니다. 델리게이트는 메서드 서명을 정의하는 유형입니다. 이를 통해 메서드를 다른 메서드에 인자로 전달할 수 있습니다. 값 대신 메서드에 대한 레퍼런스를 저장하는 변수라고 생각하면 됩니다.

반면에 이벤트는 클래스가 느슨하게 연계된 방식으로 서로 소통할 수 있게 지원하는 특수한 유형의 델리게이트입니다. 이벤트는 '패턴: 관찰자' 섹션에서 자세히 살펴볼 예정입니다.

델리게이트와 이벤트의 차이점에 대한 일반적인 정보를 알아보려면 C#의 델리게이트와 이벤트 구분(영문)을 참고하세요.

델리게이트의 기본적인 개념은 특정 작업을 수행하려면 이러한 작업을 수행하는 알고리즘을 자체 오브젝트에 캡슐화해야 한다는 것입니다. 디자인 패턴의 저자 네 명은 이 일반적인 디자인을 전략 패턴이라고 칭했습니다.

미로에서 경로를 산출하는 경로 탐색 오브젝트를 만들려고 한다고 가정해 보겠습니다. 오브젝트 자체는 경로 탐색 로직을 포함하지 않으며, 그 대신 경로를 탐색하는 다른 오브젝트에 대한 레퍼런스를 보관합니다.

특정 경로 검색 기법(A*, Dijkstra 등)을 사용하여 미로를 풀고 싶다면 이 별도의 '전략' 오브젝트 내에 올바른 솔루션을 구현하면 됩니다. 그런 다음 런타임 시에 오브젝트를 교체하여 다른 알고리즘으로 바꿀 수 있습니다.

ScriptableObject 메서드

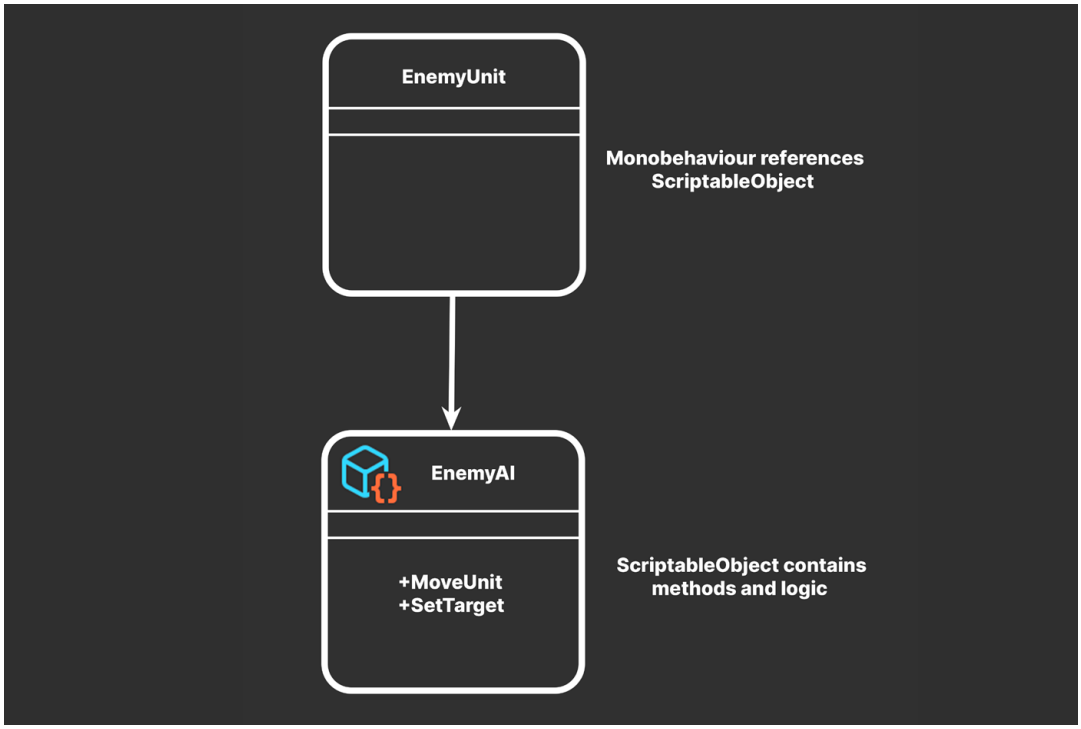
Unity에서 이 패턴을 구현하는 한 가지 방법은 필요한 로직이 들어 있는 ScriptableObject를 MonoBehaviour가 참조하도록 하는 것입니다. 그러면 MonoBehaviour가 작업을 수행할 때 자체 메서드를 호출하는 대신 ScriptableObject에 있는 외부 메서드를 호출합니다.

그러나 여기에는 몇 가지 제한이 있습니다.

— ScriptableObject의 메서드는 Start(), Update(), OnCollisionEnter() 등과 달리 MonoBehaviour의 PlayerLoop에서 자동으로 호출되지 않습니다. 따라서 직접 호출해야 합니다.

— ScriptableObject는 프리팹과 마찬가지로 씬 오브젝트를 직접 참조할 수 없습니다. 씬 오브젝트에 대해 작업을 수행해야 하는 경우, 오브젝트를 파라미터로 전달해야 합니다.

  ScriptableObject의 메서드를 호출할 때 MonoBehaviour는 보통 스스로를 인자로 전달하지만, 다른 종속성을 전달할 수도 있습니다. 따라서 ScriptableObject가 프로젝트 레벨에 있어도 유연하게 로직과 코루틴 등을 실행할 수 있습니다.



ScriptableObject는 행동이나 로직을 전환 가능하게 구현할 수 있습니다.

예를 들어 게임에서 움직임 패턴이 서로 다른 여러 개의 적 유닛을 정의했다고 가정해 보겠습니다. 순찰을 돌아야 하는 적도 있고, 경비를 서야 하는 적도 있으며, 플레이어로부터 도망쳐야 하는 적도 있을 것입니다.

이때 하나의 `EnemyUnit` MonoBehaviour가 `MoveUnit`이라는 메서드를 포함하는 `EnemyAI` ScriptableObject를 참조할 수 있습니다. `EnemyUnit` 스크립트 자체는 움직임이나 행동 로직을 포함하지 않으며, 적절한 시점에 ScriptableObject의 `MoveUnit`을 실행합니다.

메서드에 씬의 데이터가 필요한 경우, EnemyUnit 오브젝트가 자신에 대한 레퍼런스를 파라미터로 전달할 수 있습니다. 필요하다면 씬에 있는 다른 모든 종속성도 전달할 수 있습니다.

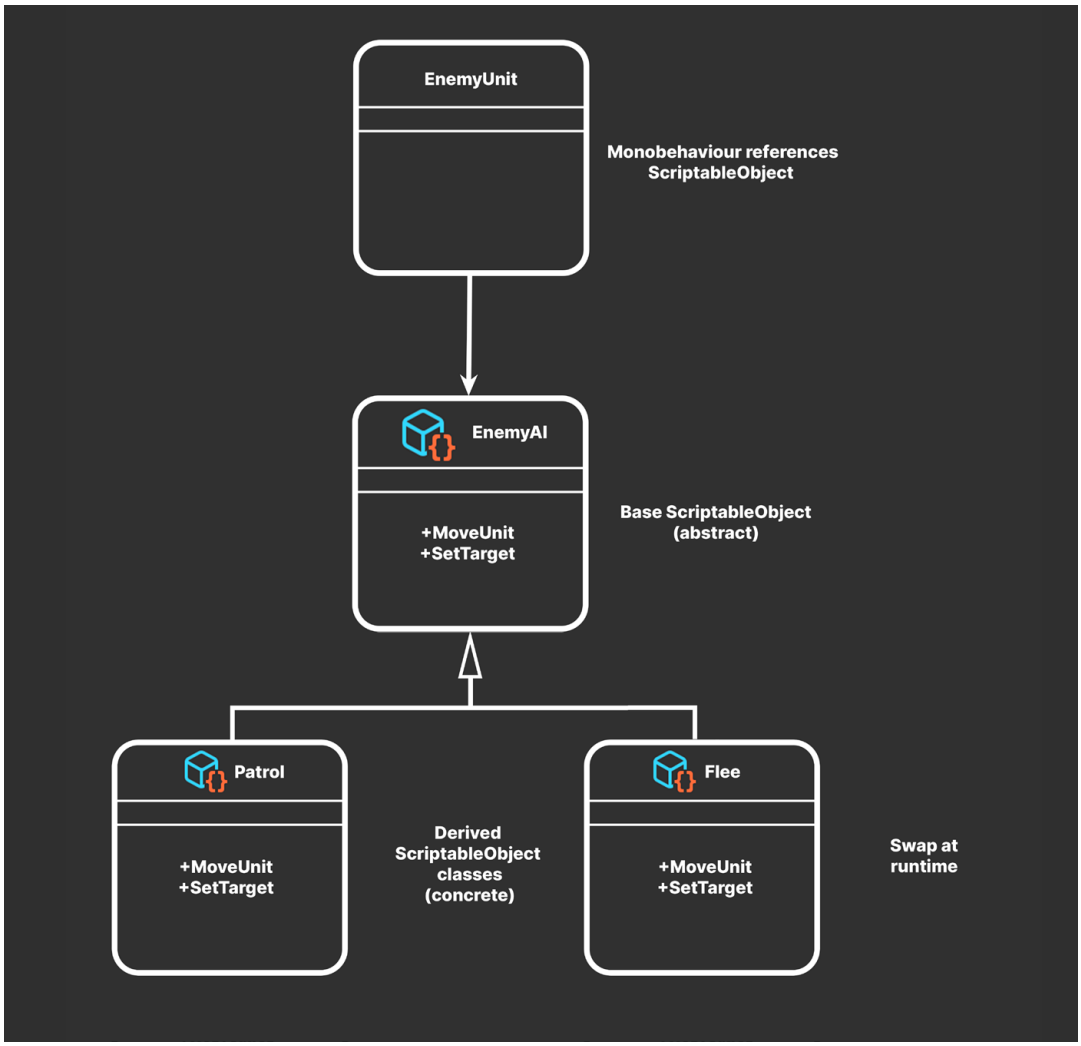
ScriptableObject 데이터 수정

런타임 중에 ScriptableObject 데이터를 변경할 수 있지만, 그렇게 할 때는 항상 주의해야 합니다. 동일한 ScriptableObject를 공유하는 여러 개의 Monobehaviour가 동일한 데이터를 수정하면 문제가 생길 수 있습니다.

런타임 중에 ScriptableObject의 인스턴스를 생성하면 이 문제를 방지할 수 있습니다. 그러면 초기 ScriptableObject가 모든 로직과 데이터를 포함하는 템플릿으로 작동합니다. 각 MonoBehaviour는 자체적으로 해당 ScriptableObject의 인스턴스를 만들게 되며, 원하는 대로 수정해도 문제가 생기지 않습니다.

전환 가능한 동작

EnemyAI ScriptableObject를 abstract 클래스로 정의하여 이 패턴을 더욱 유용하게 만들 수 있습니다. 이 `abstract` 클래스는 EnemyUnit MonoBehaviour와 호환되는 다양한 ScriptableObject의 템플릿으로 작동합니다. 즉, 추상 ScriptableObject가 둘 이상의 알고리즘의 역할을 하게 되는 것입니다.



전환 가능한 동작은 런타임 중에 또는 에디터에서 변경할 수 있습니다.

따라서 기본 EnemyAI에서 파생된 `Patrol`, `Idle`, `Flee` 같은 동작에 대한 구상 ScriptableObject 클래스를 확보할 수 있습니다. 이 클래스들은 모두 동일한 MoveUnit 메서드를 구현하지만 저마다 서로 다른 결과를 산출합니다.

에디터에서 각 에셋을 서로 교환할 수 있습니다. 원하는 ScriptableObject를 `EnemyAI` 필드로 드래그 앤 드롭하면 됩니다. 모든 호환되는 ScriptableObject는 이 방식으로 전환이 가능합니다.

EnemyUnit이나 다른 컴포넌트는 뇌처럼 작동하면서 ScriptableObject를 전환할 시점을 모니터링하며, 런타임 중에 동작을 교체할 수도 있습니다. 이는 EnemyUnit이 순찰 상태에서 도망 상태로 전환하는 등의 게임플레이 이벤트에 대응할 수 있는 한 가지 방법입니다. 각 상태 변경 시에 EnemyAI ScriptableObject를 전환하면 됩니다.

정식 제작 단계에서 다른 개발자나 디자이너가 ScriptableObject 내에 실제 움직임이나 AI 로직을 구현할 수 있습니다. 이때 게임에 DuckAndCover, Chase 등의 움직임이나 동작이 추가되어도 원본 EnemyUnit 스크립트는 변경되지 않은 채로 유지됩니다. 이 패턴 덕분에 SOLID 프로그래밍의 개방-폐쇄 원칙이 지켜지고, 결과적으로 코드베이스의 확장성이 향상됩니다. 모든 것이 이미 작은 오브젝트로 분할되어 있기 때문에 팀원이 새로 들어오거나 게임 디자인이 변경되어도 프로젝트를 더욱 원활하게 확장할 수 있습니다.

스크립터블 오브젝트를 사용하는 게임플레이 AI

스크립터블 오브젝트를 사용하여 동작을 구현하는 더 자세한 예시를 보려면 스크립터블 오브젝트와 전환 가능한 AI 동영상 시리즈(영문)를 시청하세요. 이 라이브 세션 녹화 영상에서는 상태, 동작, 상태 간 전환을 위해 스크립터블 오브젝트를 사용하여 설정할 수 있는 유한 상태 머신 기반 AI 시스템을 시연합니다.

예시: 오디오 델리게이트

스크립터블 오브젝트에 포함된 동작이 꼭 복잡해야 하는 것은 아닙니다. 커스터마이즈된 사운드 재생과 같은 기본적인 동작일 수도 있습니다.

다음은 AudioClips에 다채로움을 더해 줄 AudioDelegate 스크립터블 오브젝트의 예시입니다. `AudioDelegateSO`는 `AudioSource`를 파라미터로 받는 `Play` 메서드를 사용하여 abstract 클래스를 정의합니다.

```
using UnityEngine;
using Random = UnityEngine.Random;
using System;

[Serializable]
public struct RangedFloat
{
	public float MinValue;
	public float MaxValue;
}

public abstract class AudioDelegateSO : ScriptableObject
{
	public abstract void Play(AudioSource source);
}
```

이 구상 SimpleAudioDelegate 스크립터블 오브젝트는 사용 가능한 옵션 중에서 임의의 클립을 선택하여 재생 중에 볼륨과 피치에 변화를 줄 수 있습니다. 이렇게 하면 같은 사운드가 반복되어 생기는 단조로움이 줄어듭니다.

```
[CreateAssetMenu(fileName ="AudioDelegate")]
public class SimpleAudioDelegateSO : AudioDelegateSO
{
	public AudioClip[] Clips;
	public RangedFloat Volume;
	public RangedFloat Pitch;

	public void Play(AudioSource source)
	{
		if (clips.Length == 0 || source == null)
			return;

		source.clip = clips[Random.Range(0, Clips.Length)];
		source.volume = Random.Range(Volume.minValue, Volume.maxValue);
		source.pitch = Random.Range(Pitch.minValue, Pitch.maxValue);
		source.Play();
	}
}
```

이제 모든 MonoBehaviour가 AudioDelegateSO 클래스에서 파생된 ScriptableObject 인스턴스를 사용할 수 있습니다. 다양한 오디오 효과를 위해 여러 버전의 AudioDelegate를 만들 수도 있습니다.

이처럼 ScriptableObject에 메서드를 사용하면 다양한 가능성이 열립니다. 메서드는 단순히 작업을 수행하는 것 외에도 씬에 있는 오브젝트에 메시지를 보낼 수도 있습니다.

다음으로는 관찰자 패턴을 갖는 스크립터블 오브젝트 기반 이벤트 시스템을 살펴보겠습니다.

ScriptableObject가 불러온 혁신

이 전자책의 많은 부분은 리처드 파인이 유나이트 2016에서 선보인 MonoBehaviour의 시대에 종말을 고하는 ScriptableObject 혁명 프레젠테이션(영문)을 기반으로 작성되었습니다. 데모의 일부(원본 프로젝트에서는 AudioEvent라 칭함)는 이 예시에서 수정된 상태로 사용되었습니다.

구현 세부 정보와 ScriptableObject 사용 예시는 샘플 프로젝트를 참고하세요.

관찰자 패턴

게임을 개발할 때는 여러 개의 게임 오브젝트를 사용하며 서로 데이터나 상태를 공유해야 하는 경우가 많습니다. 규모가 작은 게임에서는 오브젝트를 직접 참조할 수 있지만, 규모가 커지면 이 방식이 효율적이지 않습니다. 이러한 종속성을 관리하려면 막대한 노력이 필요하며, 이로 인해 버그가 발생하기도 합니다.

애플리케이션의 규모가 커질수록 더 효과적인 솔루션이 필요합니다.

싱글톤 사용 지양

많은 개발자가 싱글톤, 즉 씬 로딩 후에도 존속하는 하나의 전역 인스턴스를 사용합니다. 그러나 싱글톤의 전역 상태로 인해 유닛 테스트를 하기가 까다로워집니다.

싱글톤을 참조하는 프리팹을 사용할 때는 함수 하나의 격리 테스트를 위해 모든 종속성을 임포트해야 합니다. 이는 모듈성을 저하하고 코드를 디버깅하기 어렵게 만듭니다.

그 대안으로 스크립터블 오브젝트 기반 이벤트를 사용하여 오브젝트가 서로 소통하도록 만들 수 있습니다.

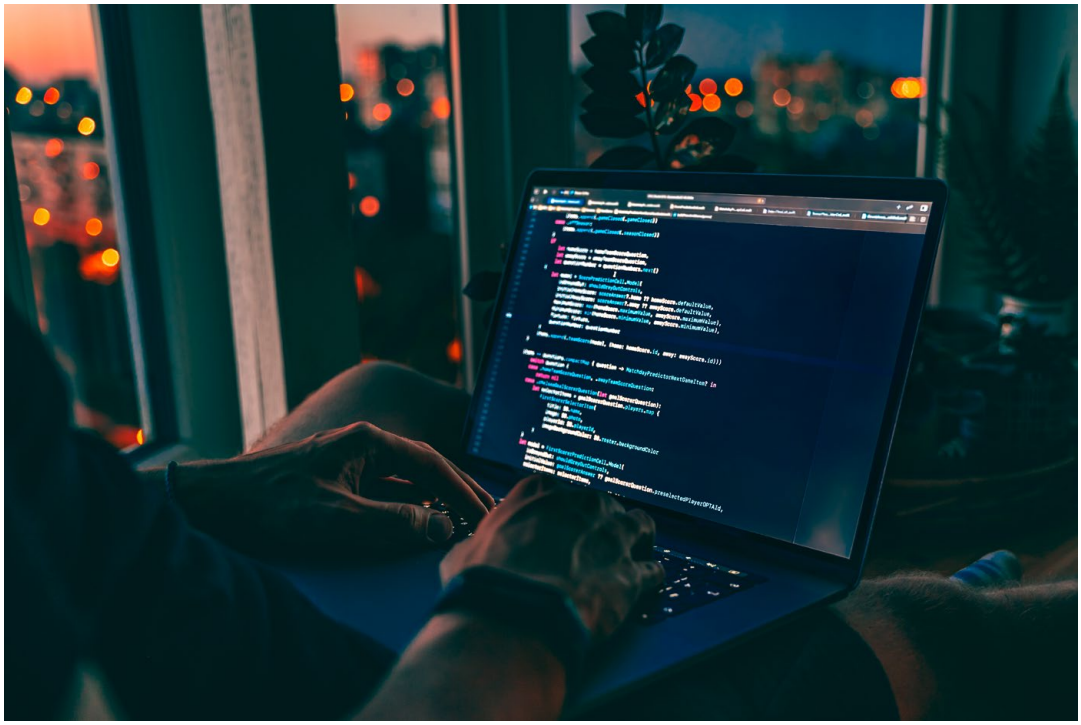
싱글톤에 대한 자세한 이야기

Unity 게임 개발에서 싱글톤이라는 주제는 치열한 토론의 대상입니다. 소규모 프로젝트나 프로토타이핑에서는 싱글톤이 적절한 해결책이 될 수 있습니다. 그러나 대규모 애플리케이션에서는 싱글톤의 단점이 장점을 덮어 버립니다. 이 때문에 많은 개발자가 싱글톤을 안티패턴으로 간주합니다.

싱글톤은 쉽게 이해하고 익힐 수 있지만 잘못 사용할 경우 문제를 유발할 수 있습니다. 여기에서 설명하는 대부분의 패턴을 활용하면 과도한 싱글톤 사용을 방지할 수 있을 것입니다.

공유 데이터에 쉽게 액세스할 방법이 필요하다면 스크립터블 오브젝트 기반 런타임 세트를 사용해 보세요(아래 참조). 오브젝트 간에 메시지를 주고받아야 한다면 스크립터블 오브젝트 기반 이벤트 채널을 활용하는 방법도 있습니다. 싱글톤을 사용하지 않도록 아키텍처를 재구성하면 확장성을 향상할 뿐 아니라 테스트를 진행하기도 더 원활해집니다.

싱글톤의 장점과 단점에 대해 자세히 알아보려면 디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드 전자책을 참고하세요.

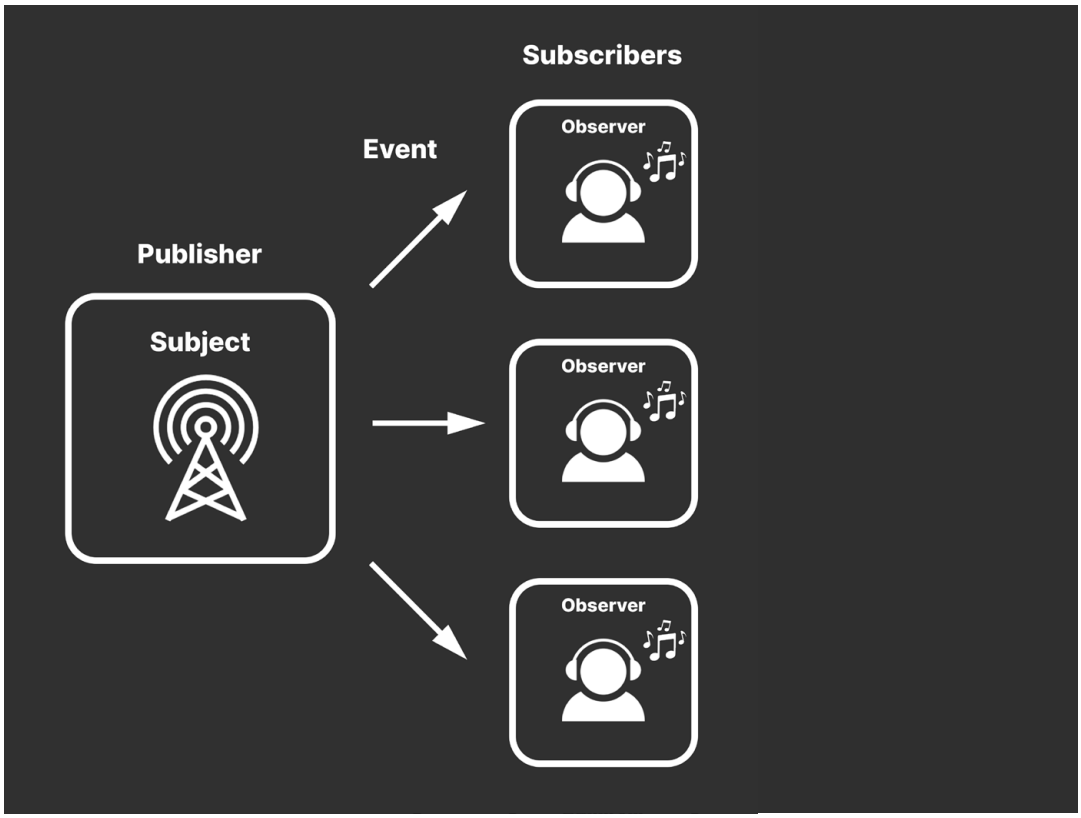


스크립터블 오브젝트 기반 이벤트

앞서 살펴보았듯이 스크립터블 오브젝트는 단순히 데이터만 다루는 것이 아니며, 다른 모든 스크립트와 마찬가지로 메서드를 포함할 수도 있습니다. 이 메서드를 오브젝트 간의 소통을 위한 수단으로 쓸 수 있습니다.

관찰자 디자인 패턴에서는 주체가 하나 이상의 느슨하게 연결된 관찰자에게 메시지를 전달합니다. 각각의 관찰 오브젝트는 주체에 의존하지 않고 독립적으로 대응할 수 있으나, 다른 관찰자들의 존재는 알지 못합니다. 주체는 '퍼블리셔' 또는 '브로드캐스터'라고도 하며, 관찰자는 '구독자' 또는 '리스너'라고도 합니다.

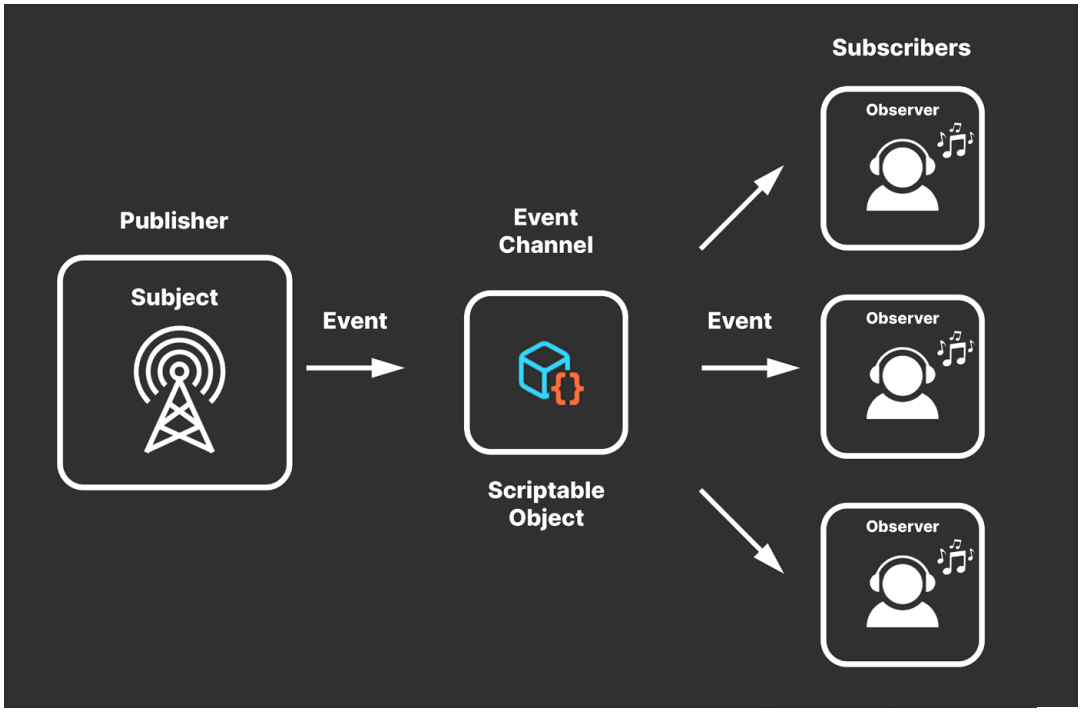
이벤트 기반 아키텍처는 프레임마다 실행되지 않고 필요할 때만 실행됩니다. 따라서 MonoBehaviour의 업데이트 메서드에 로직을 추가하는 것보다 훨씬 최적화됩니다.



기본적인 관찰자 패턴

관찰자 패턴은 MonoBehaviour 또는 C# 오브젝트를 사용하여 구현할 수 있습니다. 이는 Unity 개발 환경에서는 널리 사용되는 방식이지만, 개발 과정에서 스크립트에만 의존하면 디자이너들이 게임플레이 중에 필요한 이벤트를 구현할 때마다 프로그래밍 팀의 도움을 받아야 합니다.

스크립터블 오브젝트 기반 이벤트를 생성하면 이 문제를 극복할 수 있습니다. 디자이너에게 익숙한 방식으로 관찰자 패턴을 설정할 수 있기 때문입니다. 이 방식에서 스크립터블 오브젝트는 주체와 관찰자 사이의 중개자로 기능하여 에디터에 그래픽 인터페이스를 제공합니다.



스크립터블 오브젝트 기반 이벤트 채널

언뜻 보기에는 관찰자 패턴에 레이어가 추가되어 오버헤드가 생긴 것처럼 보이지만, 이 구조에는 몇 가지 이점이 있습니다. 스크립터블 오브젝트는 에셋이기 때문에 씬 계층 구조에 있는 모든 오브젝트가 액세스할 수 있으며 씬 로딩 시에도 사라지지 않습니다.

특정 리소스에 간편하면서도 영구적으로 액세스할 수 있다는 점은 많은 개발자가 싱글톤을 사용하는 이유이기도 합니다. 하지만 스크립터블 오브젝트는 싱글톤과 달리 불필요한 종속성을 야기하지 않으면서도 그와 동일한 이점을 제공합니다.

스크립터블 오브젝트 기반 이벤트에서는 모든 오브젝트가 (이벤트를 전달하는) 퍼블리셔로 작동할 수 있고, 모든 오브젝트가 (이벤트를 수신 대기하는) 구독자로 작동할 수 있습니다. 스크립터블 오브젝트는 중간에서 신호를 릴레이하며 둘 사이의 중앙 집중형 중개자 역할을 합니다.

일종의 '이벤트 채널'인 셈입니다. 스크립터블 오브젝트가 신호를 수신 대기하는 수많은 오브젝트로 구성된 송신탑이라고 생각해 보세요. 관심 있는 Monobehaviour는 이 이벤트 채널을 구독하기만 하면 이벤트가 발생할 때 응답할 수 있습니다.

예시: 이벤트 채널

다음을 포함하는 모든 스크립터블 오브젝트는 이벤트 채널로 작동할 수 있습니다.

— **델리게이트(UnityAction 또는 System.Action):** 구독자에 알림을 제공하고 적절한 데이터를 파라미터로 전달합니다. 아티스트 친화적인 경험을 구현하려면 UnityAction을 사용하세요. 그 밖의 경우에는 System.Action 델리게이트가 효과적입니다.

  event 키워드는 델리게이트가 ScriptableObject 클래스(또는 델리게이트가 선언되는 파생된 클래스) 내에서만 호출될 수 있도록 제한을 설정합니다.

— **이벤트 발생 메서드:** 이 public 메서드는 델리게이트를 호출합니다.

이게 전부입니다.

이벤트 채널을 원하는 개수만큼 설정하여 게임플레이의 다양한 측면을 정할 수 있습니다. 스크립터블 오브젝트는 프로젝트 레벨에 존재하기 때문에 전역적으로 액세스 가능한 이벤트를 발생시킬 수 있습니다. 덕분에 씬에서 달리 관련되어 있지 않은 오브젝트들을 확장 가능한 방식으로 연결할 수 있습니다.

System.Action 또는 UnityAction

System.Action은 .NET Framework의 System 네임스페이스 내에 정의되는 범용 델리게이트 유형으로, 커스텀 델리게이트를 선언하지 않아도 Unity 프로젝트에서 사용할 수 있습니다. event 키워드를 추가하면 델리게이트가 읽기 전용이 되며, 다른 오브젝트는 델리게이트의 등록된 메서드를 수신 대기할 수 있으나 이러한 메서드를 직접 호출할 수는 없습니다.

UnityAction은 UnityEngine.Events 네임스페이스 내에 정의되는 델리게이트 유형으로, 일반적으로 Unity에서 이벤트를 생성하는 대체 수단인 UnityEvent 클래스와 함께 사용합니다. UnityEvent와 UnityAction은 인스펙터에 표시되며, 따라서 사용자나 아티스트가 보다 쉽게 관찰자 패턴을 구현할 수 있습니다.

일반적으로는 필요에 따라 System.Action이나 UnityAction 중에 하나를 사용하며, 동일한 프로젝트에서 둘 중 하나 또는 둘 다를 배포할 수 있습니다.

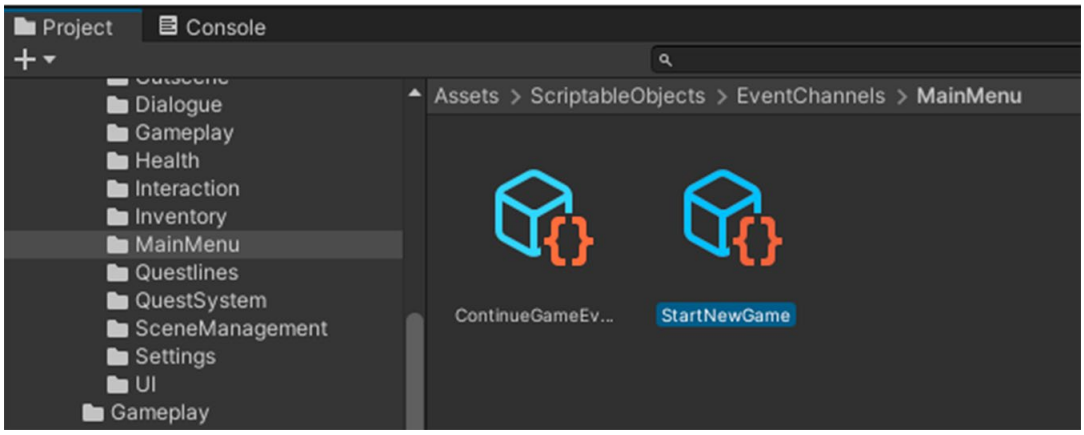
Unity 게임 엔진에 연동되지 않은 범용 델리게이트가 필요하다면 System.Action을 사용하세요. UnityEvent 전용으로 설계된 델리게이트가 필요하다면 UnityAction을 사용하세요.

파라미터 전달 없이 이벤트를 발생시키는 VoidEventChannelSO를 만들 수 있습니다. 이 클래스는 OnEventRaised라는 UnityAction을 갖습니다.

```
[CreateAssetMenu(menuName = "Events/Void Event Channel")]
public class VoidEventChannelSO : ScriptableObject
{
	public event UnityAction OnEventRaised;
	public void RaiseEvent()
	{
		if (OnEventRaised != null)
			OnEventRaised.Invoke();
	}
}
```

VoidEventChannelSO 유형의 스크립터블 오브젝트를 만들면 모든 MonoBehaviour가 OnEventRaised를 수신 대기할 수 있습니다. 예를 들어 VoidEventChannelSO 유형의 StartNewGame 스크립터블 오브젝트를 만들 수 있습니다.

다른 오브젝트가 public RaiseEvent 메서드를 호출하여 이벤트를 트리거할 수 있습니다.



이벤트 채널 예시: 스크립터블 오브젝트 기반 이벤트

다른 MonoBehaviour가 인스펙터에서 이벤트 채널 ScriptableObject를 참조하고 OnEventRaised를 구독하거나 구독을 취소할 수 있습니다.

그러면 이벤트 채널이 OnEventRaised를 호출할 때마다 그 응답으로 StartNewGame이 호출됩니다.

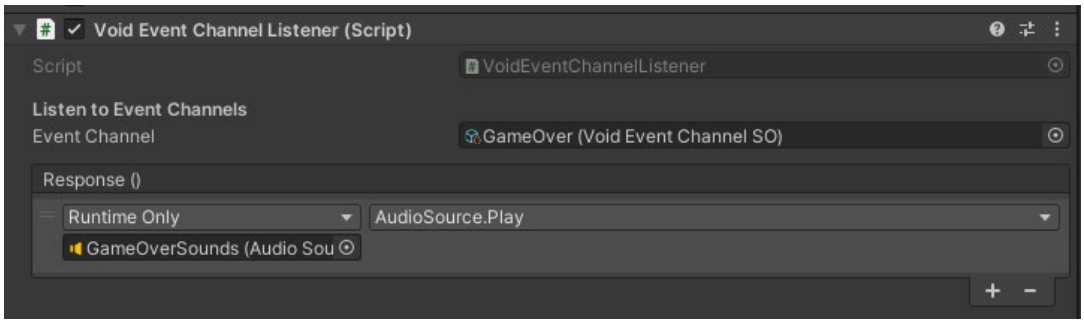
```
public class StartGame : MonoBehaviour
{
	[SerializeField] private VoidEventChannelSO m_onNewGameButton = default;

	private void Start()
	{
		m_onNewGameButton.OnEventRaised += StartNewGame;
	}

	private void OnDestroy()
	{
		m_onNewGameButton.OnEventRaised -= StartNewGame;
	}

	private void StartNewGame()
	{
	    // 여기에 레벨 로직 로드
	}
}
```

아티스트와 디자이너에게 더욱 친숙한 수신 대기 컴포넌트가 필요하다면, 스크립트 설정이 필요 없는 MonoBehaviour를 생성하는 방법이 있습니다. 아래의 VoidEventListener 클래스는 아무런 기능도 추가하지 않지만, 인스펙터에서 액세스할 수 있는 필드를 갖습니다.

```
public class VoidEventListener : MonoBehaviour
{
	[SerializeField] private VoidEventChannelSO m_channel = default;

	public UnityEvent OnEventRaised;

	private void OnEnable()
	{
		if (m_channel != null)
			m_channel.OnEventRaised += Respond;
	}
```

```
	private void OnDisable()
	{
		if (m_channel != null)
			m_channel.OnEventRaised -= Respond;
	}

	private void Respond()
	{
		if (OnEventRaised != null)
			OnEventRaised.Invoke();
	}
}
```

게임 오브젝트에 VoidEventListener를 추가하고 이벤트 채널 ScriptableObject를 인스펙터의 _channel 필드로 드래그 앤 드롭하면 됩니다. 이벤트에 응답하려면 OnRaisedEvent에 대한 UnityAction을 생성하세요.



Event Listener를 사용하면 프로그래머가 아닌 일반 사용자도 이벤트 기반 작업을 설정할 수 있습니다.

어느 컴포넌트로 이벤트를 수신 대기하든, 이벤트 채널은 런타임 시에 오브젝트들이 서로 소통할 수단을 제공합니다. 예를 들어 플레이어가 임무를 완수하거나 점수를 획득한 경우, 또는 게임이 종료된 경우 오브젝트 간에 소통이 필요합니다. 이벤트는 이 정보를 필요로 하는 씬 내 게임 오브젝트에 알림을 제공할 수 있습니다.

스크립터블 오브젝트는 프로젝트 레벨에 존재하는 에셋이므로 스크립터블 오브젝트 기반 이벤트는 애플리케이션 인프라의 많은 부분에 영향을 미칠 수 있습니다. 이는 게임 아키텍처의 기반이 되는 여러 시스템 간에 메시지를 전송할 때 특히 유용합니다.

다음은 몇 가지 일반적인 관리 시스템입니다.

— **오디오 관리:** 게임에서 사운드를 트리거할 수 있는 요소는 다양합니다. 이 시스템은 애플리케이션 이벤트에 대한 응답으로 AudioClips를 재생하거나 AudioMixer를 조정할 수 있습니다.

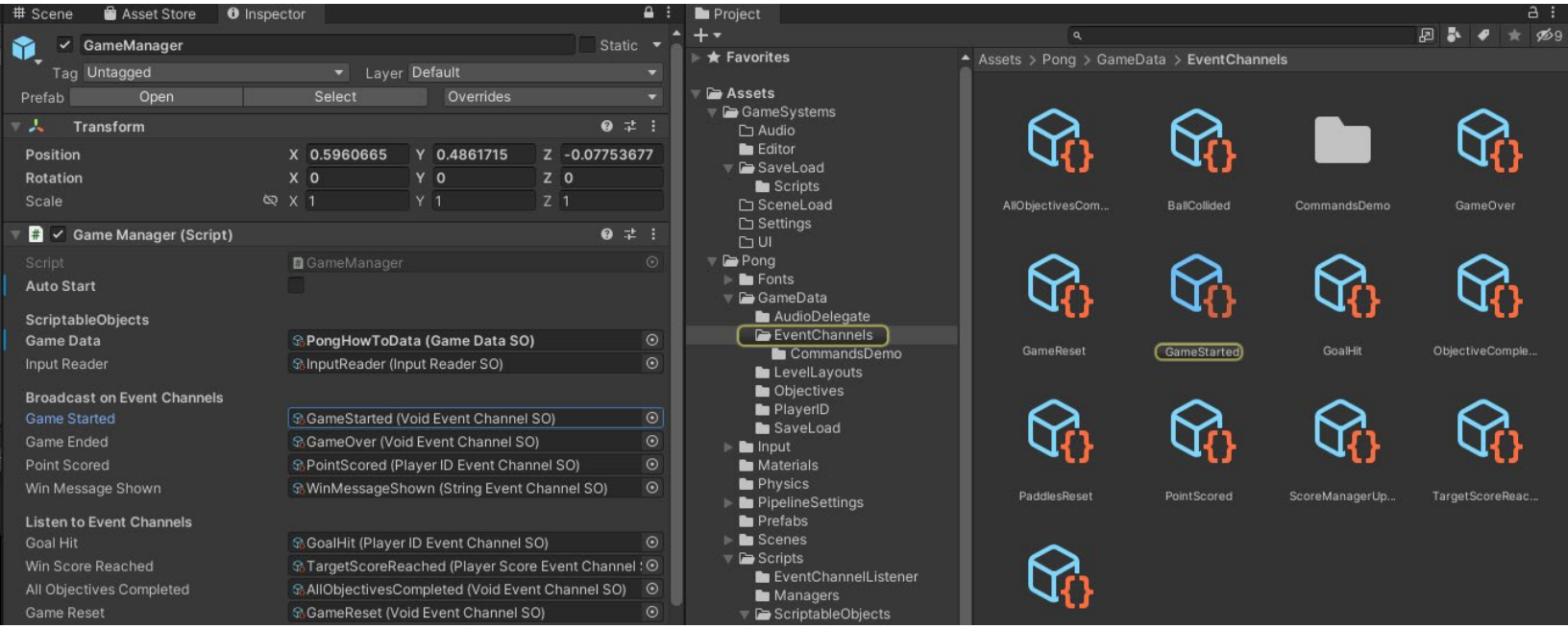
— **씬 관리**: 이 시스템은 Unity 씬과 게임 레벨의 로딩 및 언로딩을 처리합니다.

— **UI 관리:** 게임플레이 도중뿐만 아니라 전후의 메뉴 화면을 담당합니다.

— **저장 데이터 관리:** 게임 데이터의 저장, 로딩과 파일 시스템 설정을 처리합니다.

위 시스템들은 모두 서로 다른 작업을 담당하지만, 서로 소통할 수 있어야 합니다. 이때 이벤트는 이들을 연결하는 연결 고리로 작동할 수 있습니다.

스크립터블 오브젝트 기반 이벤트를 구현하는 방법을 보여 주는 더 많은 예시는 샘플 프로젝트에서 확인해 보세요.



샘플 프로젝트의 GameManager는 이벤트 채널을 사용합니다.

서로 다른 이벤트 페이로드를 사용하여 각 이벤트로 서로 다른 유형의 데이터를 보낼 수 있습니다. 예를 들어 스크립터블 오브젝트 기반 이벤트는 `IntEventChannelSO`, `Vector2EventChannelSO`, `VoidEventChannelSO` 등을 포함합니다. 사용되는 이벤트는 컨텍스트에 따라 달라집니다.

게임플레이에 따라 이벤트 유형을 추가로 커스터마이즈하세요. 이를테면 대미지 이벤트를 사용해서 대미지를 가한 주체와 대미지의 크기를 전달해야 할 수 있습니다.

이러한 이벤트 채널을 어떻게 배포할지는 여러분의 창의력에 달려 있습니다. 위에서 설명한 핵심 시스템 외에도, 이벤트를 사용하면 서로 거의 연관이 없는 인게임 시스템도 서로 상호 작용하도록 연결할 수 있습니다.

- **카메라:** 흔들림, 다른 시점으로의 전환 등 극적이거나 시네마틱한 효과를 더하기 위해 사용됩니다.
- **퀘스트:** 플레이어가 게임을 진행하거나 보상을 받으려면 완료해야 하는 작업 또는 목표입니다. 퀘스트는 아이템 가져오기, 적 퇴치하기, 퍼즐 풀기와 같은 다양한 게임플레이 요소로 구성됩니다.
- **체력:** 많은 게임의 중요한 요소로, 플레이어, 적, 그리고 플레이어에게 대미지를 가할 수 있는 오브젝트나 동작을 연결합니다.
- **업적:** 퀘스트와 마찬가지로 플레이어가 게임 내에서 특정 작업이나 목표를 완료하여 얻을 수 있는 특별한 보상입니다. 업적은 특정 레벨 달성, 특정 점수 획득 등 다양한 게임플레이 요소를 포괄할 수 있습니다.

이러한 게임플레이 요소는 이벤트를 사용하여 오디오, UI, 저장 데이터를 비롯한 다른 관리 시스템과 상호 작용합니다. 이 접근 방식을 활용하면 아키텍처 내에서 각 컴포넌트의 모듈성과 독립성을 증진하는 동시에 서로 다른 시스템 간의 소통을 지원할 수 있습니다.

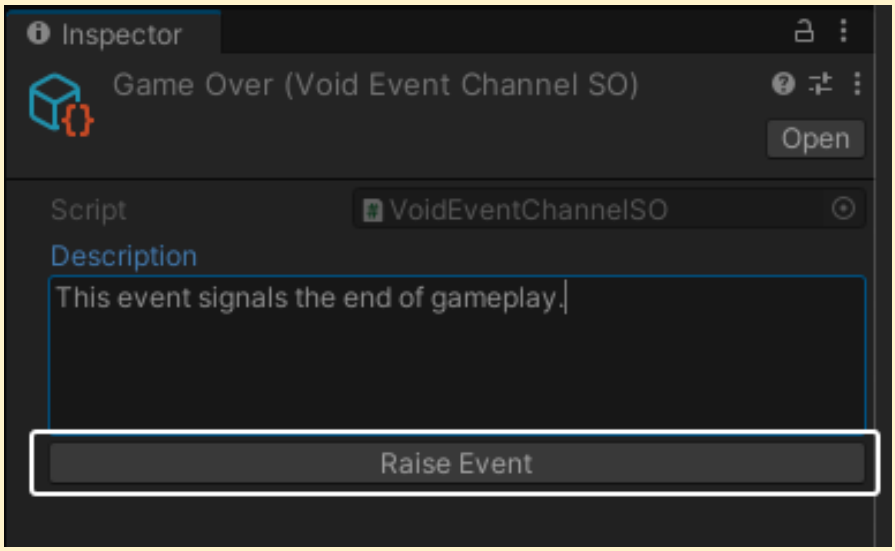
이벤트 채널 디버깅

커스텀 에디터 또는 프로퍼티 드로어로 인스펙터에 Raise Event 버튼을 생성할 수 있습니다. 이는 개발자가 이벤트를 수동으로 호출하여 디버깅하는 데 유용합니다.

예를 들어 다음은 VoidEventChannelSO에 대한 커스텀 인스펙터 버튼을 생성하는 기본적인 에디터 스크립트입니다.

```
[CustomEditor(typeof(VoidEventChannelSO))]
public class VoidEventChannelSOEditor : Editor
{
    public override void OnInspectorGUI()
    {
        DrawDefaultInspector();

        VoidEventChannelSO eventChannel = (VoidEventChannelSO)target;

        if (GUILayout.Button("Raise Event"))
        {
            eventChannel.RaiseEvent();
        }
    }
}
```

위 스크립트는 원할 때마다 이벤트를 발생시키는 버튼을 생성하며, 이를 통해 런타임 중에 쉽게 문제를 진단할 수 있습니다.



커스텀 에디터 버튼은 이벤트 채널을 테스트하는 데 유용합니다.

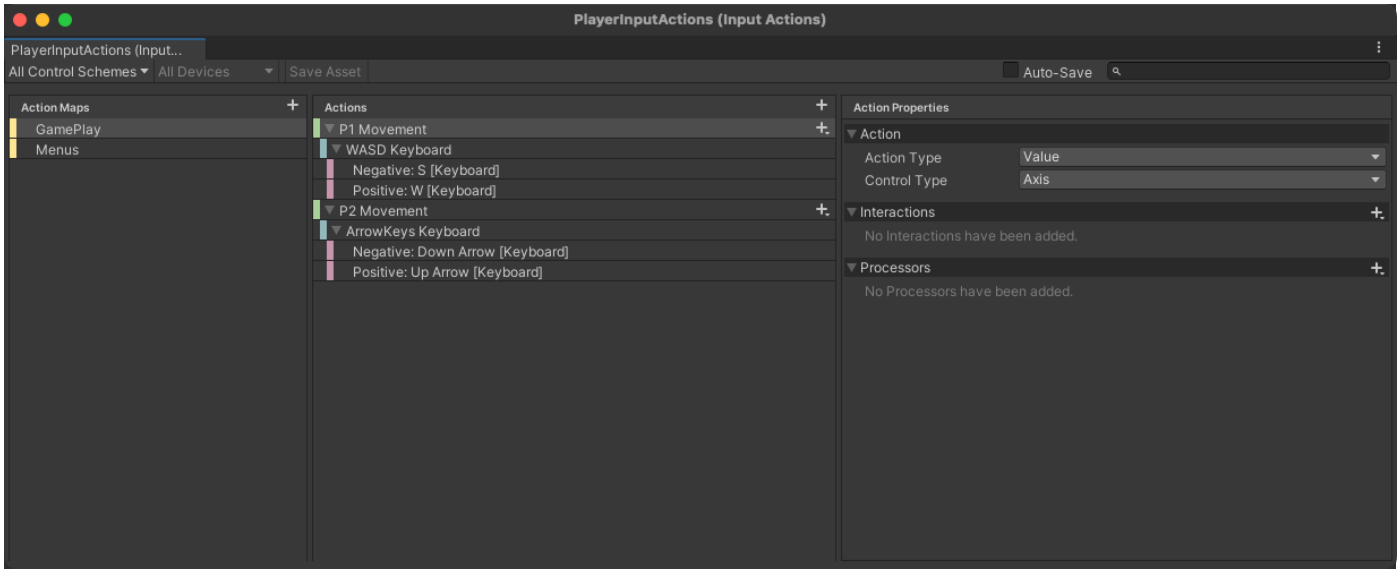
약간의 수고를 더하면 데이터를 포함하는 이벤트 채널 버튼을 만들 수도 있습니다. 이벤트 채널 자체에 디버그 값을 위한 필드를 예약하고, 그 필드를 에디터 스크립트로 전달하세요. SOAP 프로젝트와 Unity Atoms 프로젝트에 구현 방법의 예시가 자세히 나와 있습니다.

오브젝트 분리를 위해 이벤트 채널을 사용하는 과정에서, 각 이벤트의 모든 리스너를 기록하는 등의 디버깅 툴을 자체적으로 개발하는 것이 좋습니다. 이벤트 채널 클래스에 오브젝트를 구독 및 구독 취소하는 메서드를 포함하면 런타임 중에 어떤 이벤트가 특정 동작을 유발하는지 쉽게 파악할 수 있습니다.

예시: InputReader

사용자 입력을 수신 대기하는 오브젝트에는 특수한 유형의 이벤트 채널이 필요합니다. Unity의 Input System은 InputAction을 사용하여 원시 입력 데이터를 논리적인 개념으로 표현합니다(예: jump, walk).

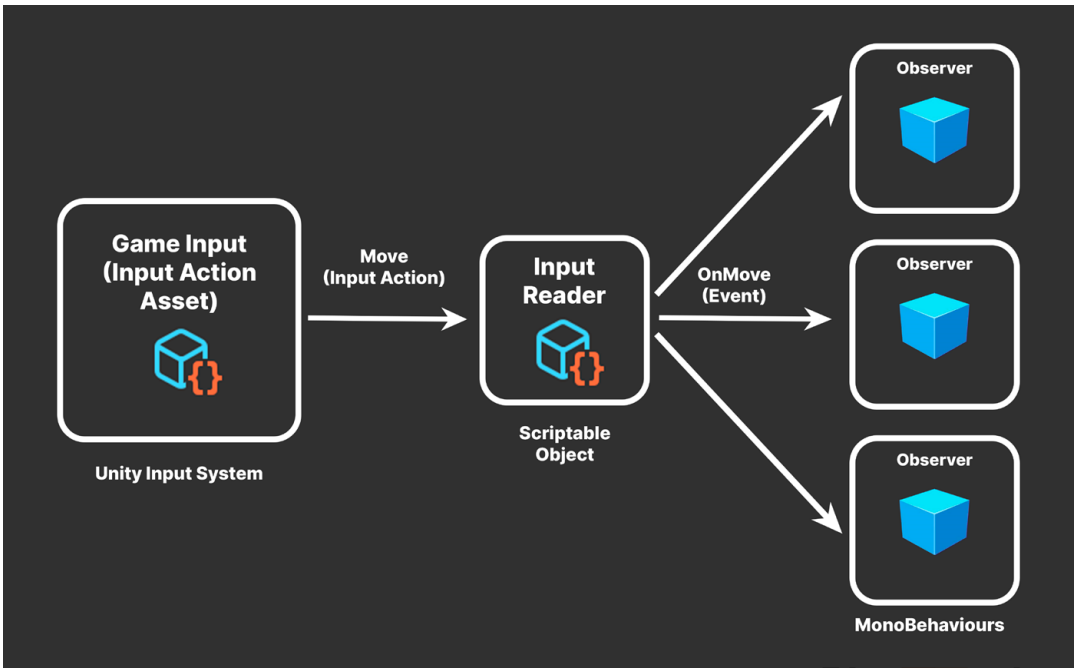
각 InputAction은 저마다 `started`, `performed`, `canceled` 이벤트를 갖습니다.



Input System 에디터에서 Actions 및 Action Maps 설정

InputAction 바인딩을 해독하기 위해 특수 InputReader ScriptableObject를 생성할 수 있습니다. 이는 주체와 관찰자 사이의 중개자로 작동합니다. 단, 이 경우에는 MonoBehaviour가 이벤트를 명시적으로 발생시키지 않습니다.

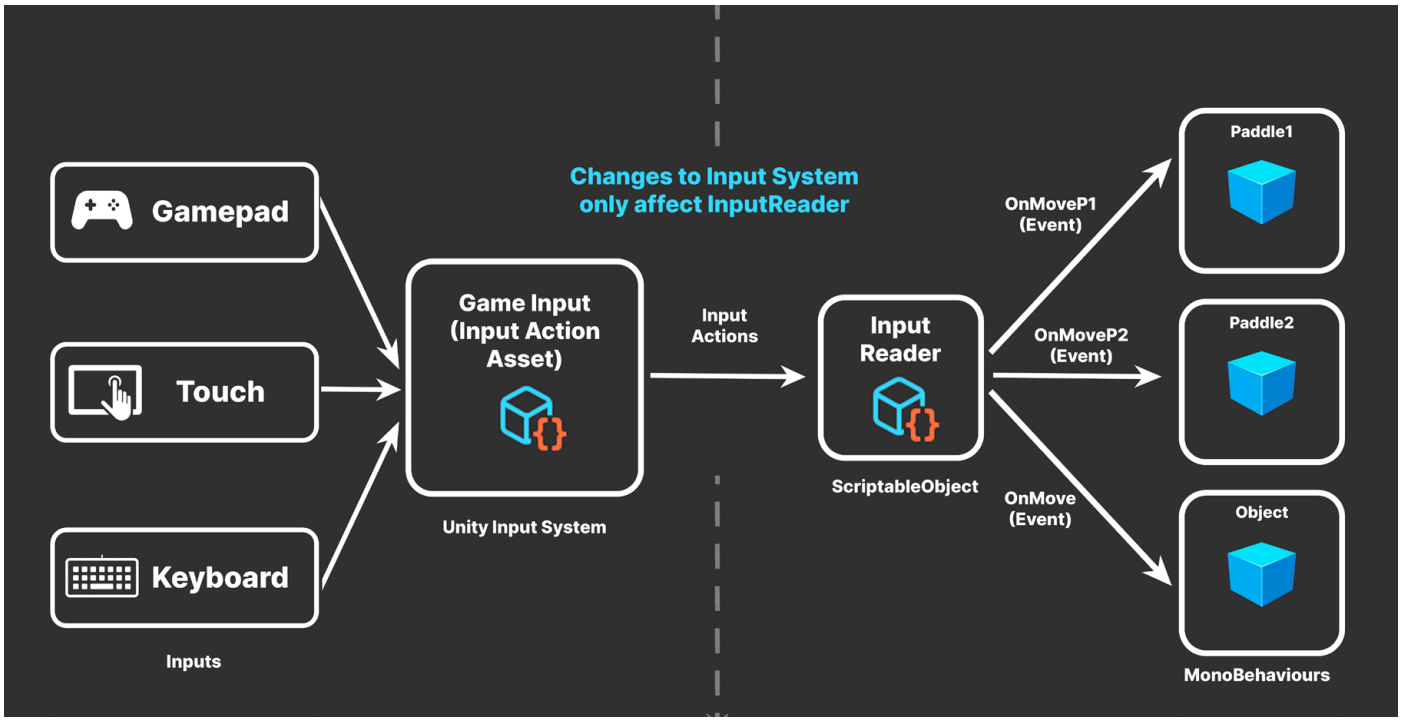
그 대신 Input System이 주체, 즉 브로드캐스터를 대신합니다.



ScriptableObject InputReader는 InputAction의 이벤트를 릴레이합니다.

여기서는 Input System에서 Actions와 ActionMaps를 설정합니다. 각 InputAction은 별도의 입력축을 기술하며, 키보드, 게임패드 또는 개발자가 지정하는 입력 기기에 바인드됩니다.

패들 컨트롤러는 InputAction을 직접 구독하는 대신 각각 `OnMoveP1.performed` 이벤트와 `OnMoveP2.performed` 이벤트를 수신 대기합니다.



InputReader는 오브젝트가 입력에 직접적으로 종속되지 않도록 합니다.

최종 InputReader는 게임 오브젝트가 게임패드 또는 키보드 동작을 처리하는 방식을 표준화합니다. 입력이 필요한 모든 게임 오브젝트는 다음과 같은 특성을 가집니다.

— InputReader ScriptableObject에 대한 레퍼런스를 보유합니다.

— 관련 이벤트를 구독하고 이벤트 처리 메서드를 연결합니다.

이 패턴은 규모가 작은 게임에는 과할 수도 있지만, 이해를 돕기 위해 이 개념을 설명했습니다.

프로젝트 규모가 커지고 다량의 컴포넌트가 추가되기 전까지는 장점이 명확히 눈에 띄지 않을 수 있습니다. 입력을 사용하는 게임 오브젝트로부터 입력을 분리하면 유연성과 재사용성이 향상됩니다.

개발 중에 InputAction을 수정해야 한다면 InputReader만 수정하면 되며, 이벤트를 변경할 필요가 없다면 수신 대기하는 오브젝트는 영향을 받지 않습니다. 따라서 입력과 관찰자 사이의 연결을 관리하기가 수월해지고, 관찰자가 많을수록 그 이점은 더 뚜렷해집니다.

정적 이벤트와 비정적 이벤트

수신 대기하는 오브젝트에서 ScriptableObject를 쉽게 찾을 수 있도록 `static` 이벤트를 사용할 수 있습니다.

예를 들어 다음과 같이 MonoBehaviour가 `OnEnable` 메서드에서 InputReader의 정적 MoveP1Event 이벤트와 MoveP2Event 이벤트를 구독할 수 있습니다.

```
InputReader.MoveP1Event += OnMoveP1;

InputReader.MoveP2Event += OnMoveP2;
```

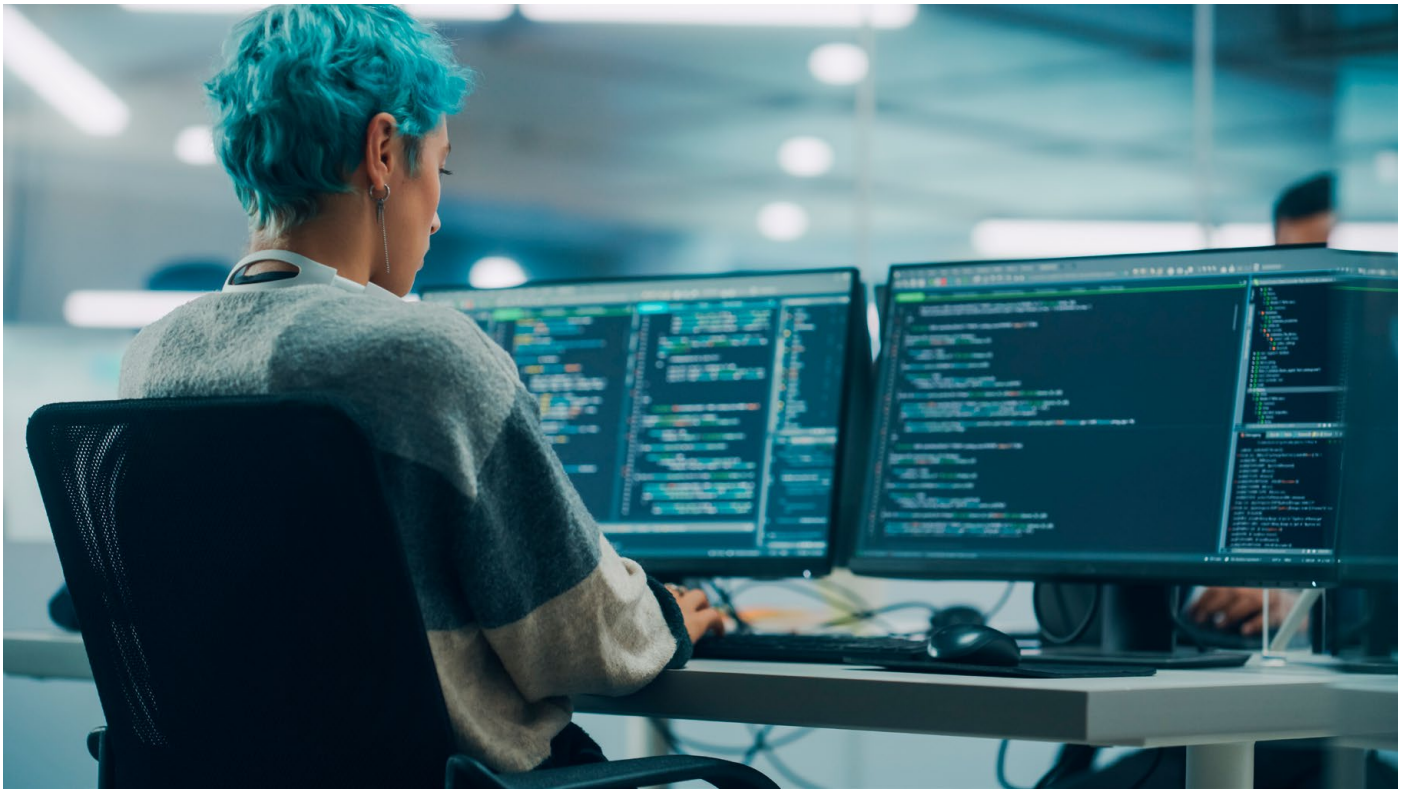
정적 이벤트를 사용할 때는 구독 관리에 신경을 써야 합니다. `OnDisable`에서 구독을 취소하는 것을 잊지 마세요.

```
InputReader.MoveP1Event -= OnMoveP1;

InputReader.MoveP2Event -= OnMoveP2;
```

정적 이벤트는 언제나 접근할 수 있으며, 활성 구독자가 있으면 가비지 컬렉터에 의해 수집되지 않습니다. 구독자가 하나라도 남아 있으면 애플리케이션이 실행되는 동안 클린업이 수행되지 않습니다.

정적 이벤트는 직렬화할 수 없습니다. 에디터에서 인터랙티브 방식으로 작업하려면 비정적 이벤트를 선택하고 인스펙터에서 적절한 ScriptableObject를 참조해야 합니다.

커맨드 패턴



메서드를 직접 호출하는 대신 커맨드 패턴을 사용하면 '커맨드 오브젝트'라는 하나 이상의 메서드 호출을 캡슐화할 수 있습니다.

그런 다음 커맨드 오브젝트를 대기열이나 스택과 같은 컬렉션에 저장할 수 있으며, 컬렉션은 작은 버퍼처럼 작동합니다. 이렇게 하면 각 커맨드 오브젝트의 실행을 유연하게 제어할 수 있습니다. 커맨드 패턴은 특정 시점에 일련의 동작을 재생하거나 해당 동작을 수행할 수 없게 만드는 등의 용도로 쓰입니다.

여러분이 좋아하는 게임 장르에서 커맨드 패턴을 경험해 본 적이 있을 것입니다.

— 실시간 전략 게임을 예로 들면 유닛과 건물의 동작을 대기열에 대기시키는 용도로 커맨드 패턴을 사용할 수 있습니다. 게임은 자원이 확보되면 건설 커맨드를 실행하거나, 혹은 유닛의 이동처럼 일련의 행동을 순차적으로 실행하게 됩니다.

— 턴제 전략 게임의 경우 플레이어가 유닛을 선택하면 이동이나 동작을 대기열이나 다른 컬렉션에 저장할 수 있습니다. 그런 다음 턴을 마치는 시점에 게임이 플레이어의 대기열에 있는 모든 커맨드를 실행합니다.

— 퍼즐 게임에서는 커맨드 패턴을 사용하면 플레이어가 동작을 실행 취소하거나 재실행할 수 있습니다.

— 격투 게임에서는 특정 커맨드 목록에 있는 버튼 입력이나 게임패드 모션을 읽으면 콤보와 필살기를 수행할 수 있습니다.

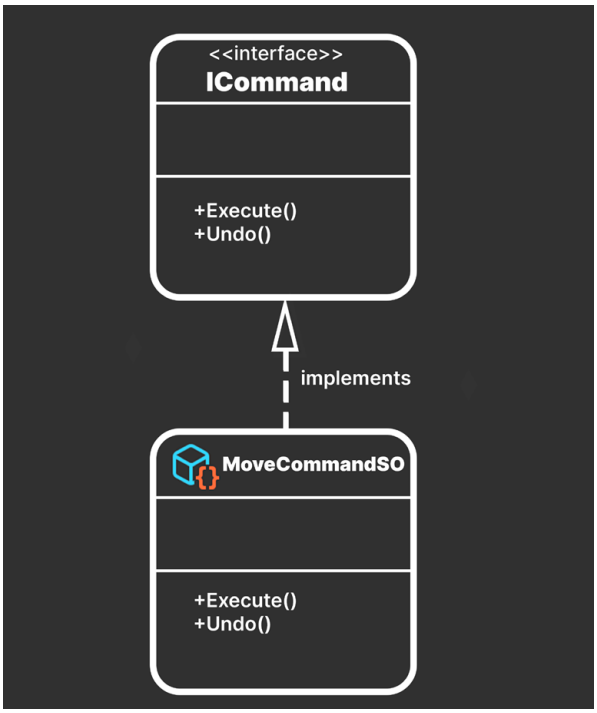
ScriptableObject를 사용하면 커맨드 패턴을 구현할 수 있습니다.

예를 들면 트랜스폼의 움직임을 정의하는 커맨드를 만드는 것입니다. 각 동작을 별도의 오브젝트로 래핑하면 더욱 정교하게 제어할 수 있습니다.

여기에서는 Execute 메서드와 Undo 메서드를 갖는 interface ICommand를 정의합니다(abstract 클래스를 사용할 수도 있음).

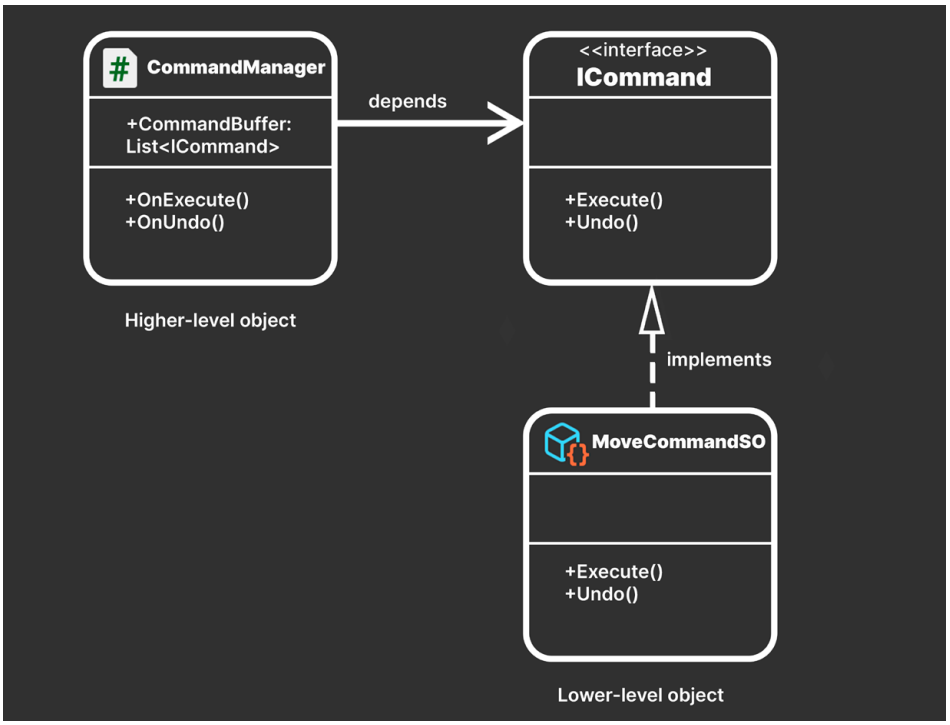
```
public interface ICommand
{
    public abstract void Execute();
    public abstract void Undo();
}
```

그런 다음 ScriptableObject로 ICommand 인터페이스를 구현합니다. 각 Command 오브젝트는 자신의 Execute 메서드와 Undo 메서드를 자체적인 구현 세부 정보로 채웁니다.



ScriptableObject를 사용하여 커맨드 패턴 구현

그런 다음 MonoBehaviour 또는 ScriptableObject가 커맨드 버퍼를 정의할 수 있습니다. 이 버퍼에는 커맨드 오브젝트가 포함됩니다. 버퍼는 목록, 스택, 배열, 대기열과 같은 컬렉션일 수 있습니다.



CommandManager는 ICommand 오브젝트의 컬렉션을 관리합니다.

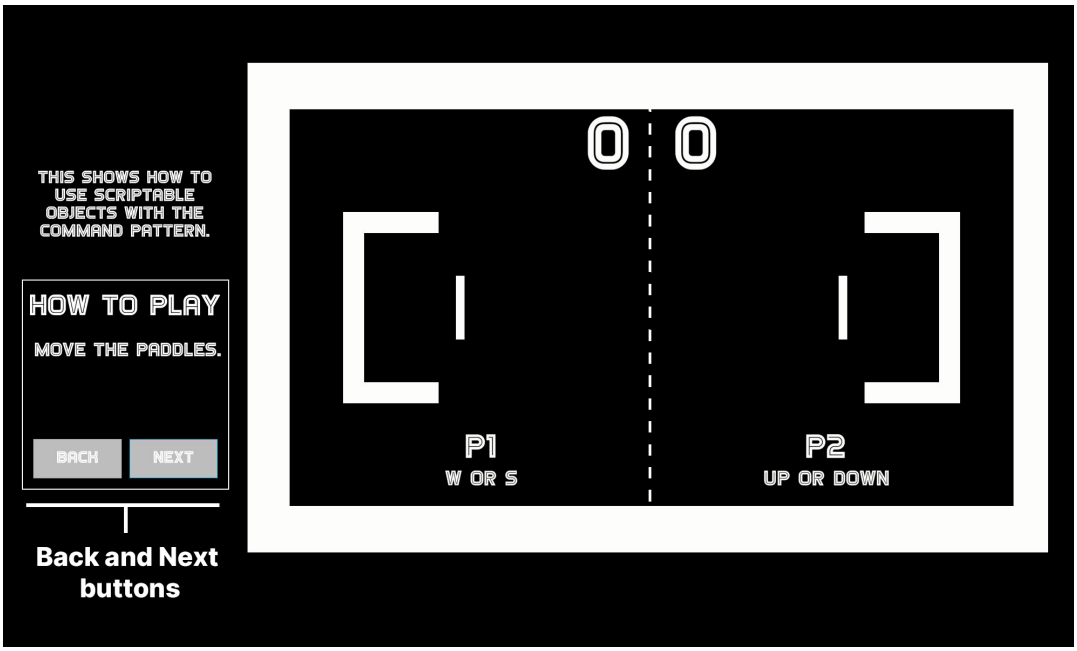
이 간단한 구조를 사용하여 커맨드를 순서대로 실행할 수 있습니다. 게임 오브젝트가 지정된 동작 또는 애니메이션을 수행해야 하는 튜토리얼이나 컷씬이 있다고 생각해 보세요. 커맨드 패턴은 이러한 용도에 적합합니다.

각 command는 별도의 오브젝트이기 때문에 순서를 변경하기도 쉽습니다. CommandBuffer를 어떻게 관리할지만 정하면 됩니다.

- 스택으로 만드는 경우에는 커맨드를 실행할 때 스택으로 커맨드를 푸시합니다. 동작을 실행 취소할 때는 재실행 스택을 별도로 관리할 수 있습니다.
- 목록이나 배열을 사용할 때는 현재 커맨드의 인덱스를 추적하고 커맨드의 실행 취소 또는 재실행에 따라 인덱스의 값을 높이거나 낮춥니다.

실행 취소 가능한 움직임의 예시는 샘플 프로젝트의 MoveCommandSO 클래스와 CommandManager 클래스를 참고하세요. 이 예시에서는 단순한 튜토리얼 씬을 통해 게임 보드의 여러 부분에 레이블을 표시하고 플레이 방법을 설명합니다.

Next 버튼을 클릭하여 설명 텍스트를 하나씩 넘깁니다. 이전 설명으로 돌아가려면 Back 버튼을 클릭합니다.



커맨드 패턴의 간단한 구현 예시

디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드 전자책에서 커맨드 패턴에 대해 더 자세히 알아보세요. ScriptableObject를 사용한 커맨드 패턴을 보여 주는 ScriptableObject와 커맨드 패턴 커뮤니티 게시물(영문)도 도움이 될 것입니다.

스크립터블 오브젝트와 C# 클래스

프로젝트에 어떤 코드 아키텍처를 사용할지 정할 때는 팀원들의 기술, 선호도와 더불어 게임의 성능 요구 사항을 고려해야 합니다. 어떤 디자이너는 Unity 에디터를 인터랙티브하게 사용하는 방식을 선호할 수도 있고, 또 어떤 디자이너는 전적으로 C# 코드로 작업하는 방식을 선호할 수도 있습니다.

모든 팀원이 쉽게 작업할 수 있는 코드베이스를 만들려면 이러한 요인을 고려하세요. 물론 그 어떤 디자인 패턴도 모든 상황에 맞는 만능 해결책이 될 수는 없습니다. 따라서 실제로 구현하기에 앞서 각 디자인 패턴의 장점과 단점을 꼼꼼히 살펴봐야 합니다.

여기에서 '적절한' 코드 아키텍처는 팀과 프로젝트에 알맞은 아키텍처라는 사실을 기억하세요.

런타임 세트 패턴

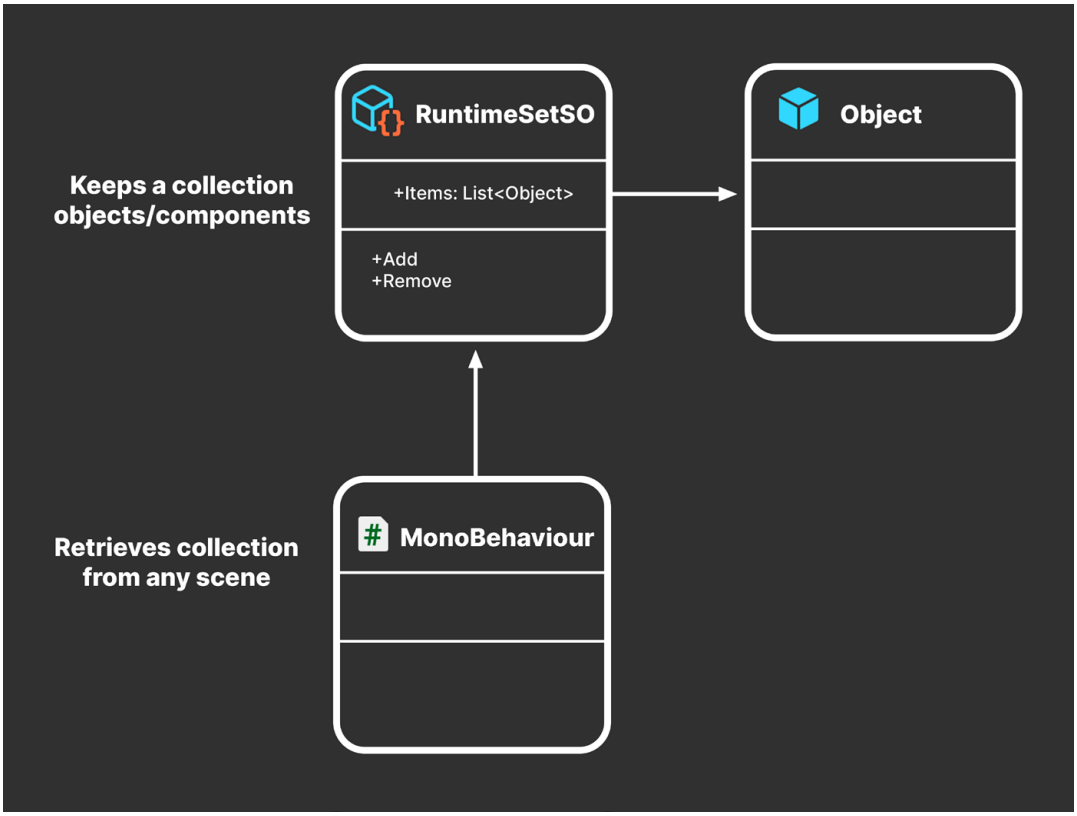
런타임 중에 게임 오브젝트의 목록이나 씬의 컴포넌트를 추적해야 하는 경우가 많습니다. 예를 들면 적 또는 NPC 목록을 관리해야 할 수 있습니다.

ScriptableObject 인스턴스는 프로젝트 레벨에서 나타나므로, 이 인스턴스에 저장된 데이터는 모든 씬의 모든 오브젝트가 사용할 수 있습니다. 이렇게 하면 싱글톤의 단점은 없애면서 간편한 전역 액세스라는 장점만 활용하는 것입니다.

이에 더해 ScriptableObject에서 직접 데이터를 읽는 것은 `Object.FindObjectOfType`이나 `GameObject.FindWithTag` 같은 찾기 연산으로 씬 계층 구조를 검색하는 것보다 효율적입니다. 사용 사례와 계층 구조의 크기에 따라 프레임별 업데이트에 찾기 연산 메서드를 사용하면 상대적으로 많은 리소스가 소모되어 비효율적일 수 있습니다.

기본적인 런타임 세트

대신 데이터를 ScriptableObject에 런타임 세트로 저장하는 방안을 고려해 보세요. 런타임 세트는 요소들의 퍼블릭 컬렉션을 관리하는 특수 데이터 컨테이너로, 컬렉션을 추가하고 삭제하는 기본적인 메서드도 제공합니다.



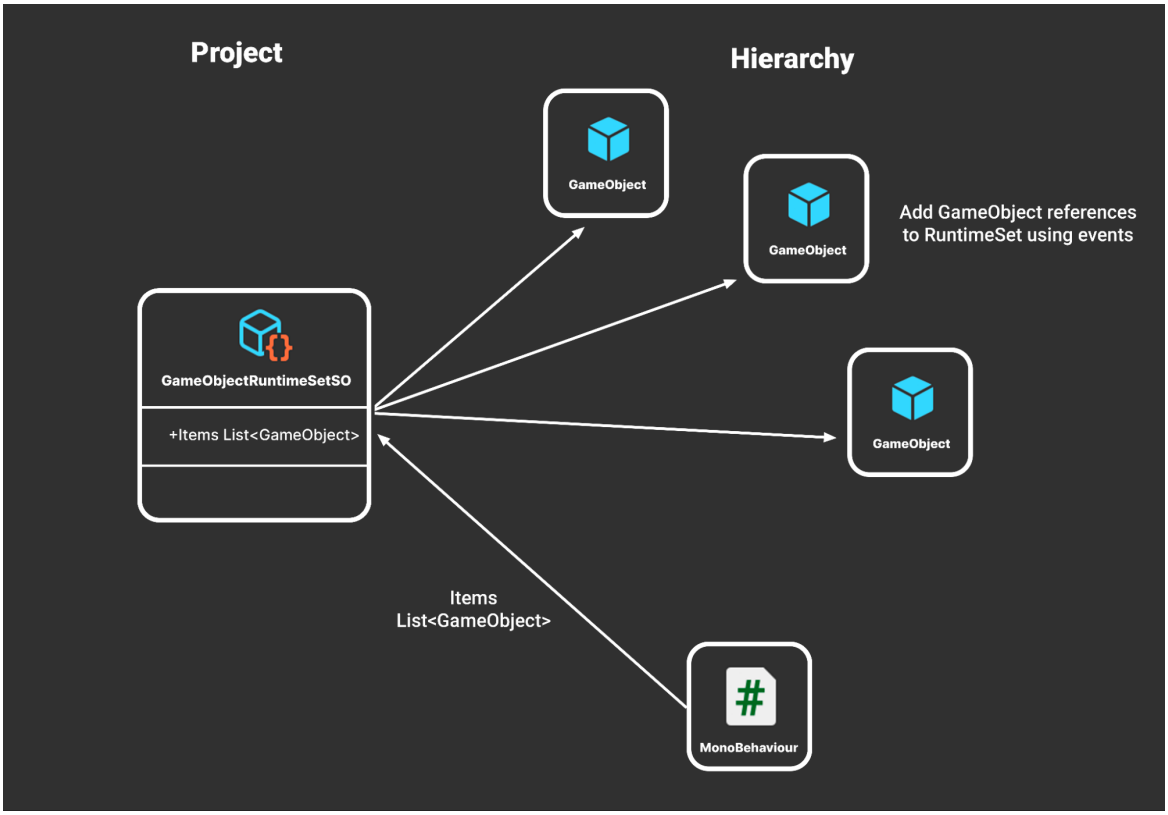
런타임 세트는 데이터 컬렉션에 대한 전역 액세스를 제공합니다.

다음은 게임 오브젝트 목록을 추적하는 기본적인 런타임 세트입니다.

```
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "GameObject Runtime Set", fileName = “GORun-
timeSet”)]
public class GameObjectRuntimeSetSO : ScriptableObject
{
    private List<GameObject> items = new List<GameObject>();
    public List<GameObject> Items => items;

    public void Add(GameObject thingToAdd)
    {
        if (!items.Contains(thingToAdd))
            items.Add(thingToAdd);
    }
```

```
    public void Remove(GameObject thingToRemove)
    {
        if (items.Contains(thingToRemove))
            items.Remove(thingToRemove);
    }
}
```

런타임 시에 모든 MonoBehaviour가 public `Items` 프로퍼티를 참조하여 전체 목록을 얻을 수 있습니다. 게임 오브젝트가 이 목록에 추가되거나 삭제되는 방식을 관리하려면 다른 스크립트 또는 컴포넌트가 필요합니다.



게임 오브젝트 런타임 세트

MonoBehaviour에서 런타임 세트를 참조합니다. 그런 다음 `OnEnable`과 `OnDisable` 이벤트 함수를 통해 런타임 세트의 `Items` 목록에서 오브젝트를 추가하거나 삭제합니다. 또는 이벤트 채널을 사용하여 게임 오브젝트를 페이로드로 전송합니다(예: GameObjectEventChannel).

제네릭 버전

런타임 세트를 특정 유형의 MonoBehaviour와 함께 사용해야 하는 상황이 있습니다. 이렇게 하면 예를 들어 씬에 있는 모든 항목이 액세스할 수 있는 적 또는 픽업 아이템 목록을 관리할 수 있습니다. 이때 유형별로 특정 런타임 세트를 만들 수 있습니다(예: EnemyRuntimeSet, PickupRuntimeSet).

추가 런타임 세트를 쉽게 만드는 한 가지 방법은 제네릭 abstract 클래스를 사용하는 것입니다.

```
public abstract class RuntimeSetSO<T> : ScriptableObject
 {
     [HideInInspector]
     public List<T> Items = new List<T>();

     public void Add(T thing)
     {
         if (!Items.Contains(thing))
             Items.Add(thing);
     }

     public void Remove(T thing)
     {
         if (Items.Contains(thing))
             Items.Remove(thing);
     }
 }
```

이 클래스는 원본 GameObjectRuntimeSet와 비슷하지만 더 유연하게 사용할 수 있습니다. 커스텀 Foo 컴포넌트를 위한 런타임 세트를 만들려면 다음과 같이 구상 FooRuntimeSetSO를 생성합니다.

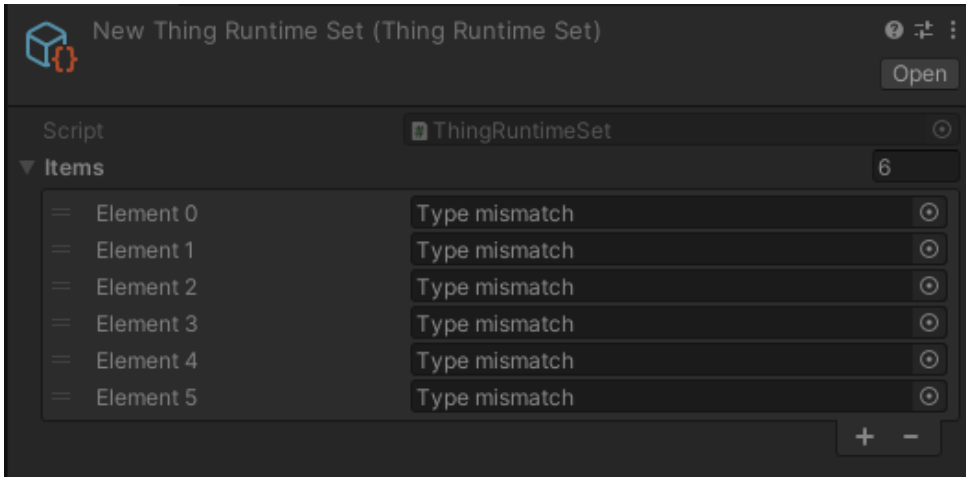
```
 [CreateAssetMenu(menuName = "Foo Runtime Set", fileName = "FooRuntime-
Set")]
  public class FooRuntimeSetSO : RuntimeSet<Foo>
  {
  }
```

게임플레이에 필요한 만큼 구상 클래스를 만드세요. 이를테면 적, NPC, 인벤토리 아이템, 퀘스트 등이 각각 런타임 세트를 보유할 수도 있습니다. 올바른 유형으로 새로운 빈 클래스를 선언하기만 하면 됩니다.

이벤트를 사용하는 대신 각 Foo 컴포넌트가 `OnEnable` 또는 `OnDisable` 메서드를 사용하여 스스로를 추가하거나 삭제하는 방법도 있습니다. 그러고 나서 인스펙터에서 FooRuntimeSet 필드를 설정하면 Foo 컴포넌트가 자동으로 런타임 세트에 나타납니다. 이는 Foo 컴포넌트를 프리팹과 함께 사용하는 경우에 특히 유용합니다.

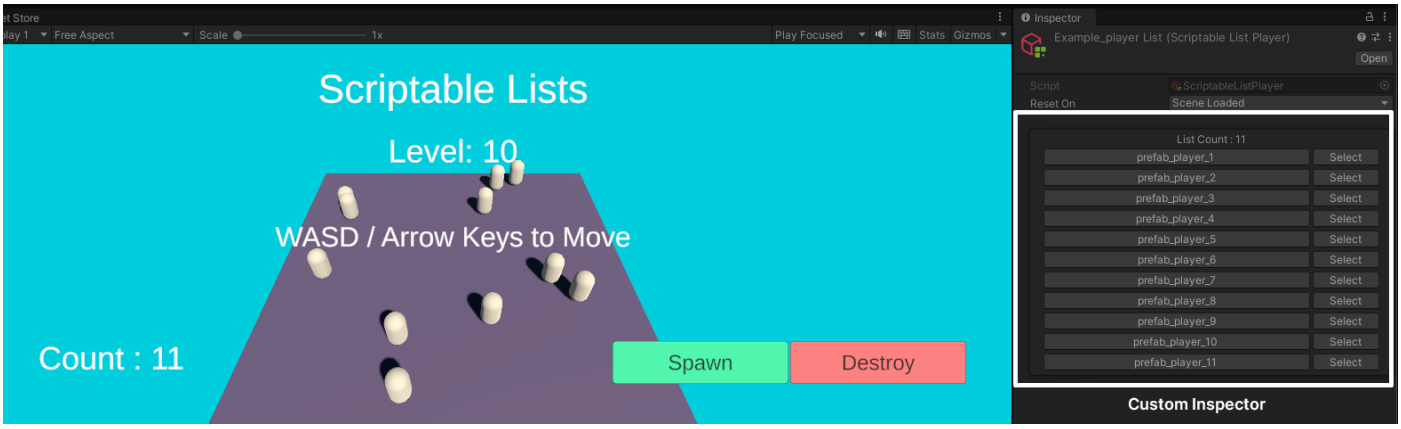
```
public class Foo : MonoBehaviour
 {
       public FooRuntimeSetSO RuntimeSet;

       private void OnEnable()
       {
           RuntimeSet.Add(this);
       }

       private void OnDisable()
       {
           RuntimeSet.Remove(this);
       }
 }
```

참고: 이 기법의 한 가지 제약은 런타임 중에 스크립터블 오브젝트를 확인해도 인스펙터에서 런타임 세트 목록의 내용을 볼 수 없다는 것입니다. 인스펙터에 목록을 공개적으로 노출하려고 시도하면 다음과 같은 화면이 표시됩니다.



런타임 세트는 인스펙터에 씬 오브젝트나 컴포넌트를 표시하지 않습니다.

스크립터블 오브젝트는 씬 오브젝트를 직렬화할 수 없으므로 기본적으로 각 요소 필드에 'Type mismatch'라는 메시지가 표시됩니다. 목록은 정상적으로 작동하지만 데이터는 올바르게 표시되지 않는 것입니다. 혼동을 방지하고 목록이 인스펙터에 표시되지 않도록 하려면 public 프로퍼티나 HideInInspector 속성을 사용하세요. 커스텀 에디터 스크립트와 인스펙터를 사용하여 이 문제를 해결할 수도 있습니다. 이 문제에 대한 예시로 에셋 스토어의 SOAP(Scriptable Object 아키텍처 패턴)을 참고하세요.



SOAP의 커스텀 에디터에는 런타임 세트의 내용이 표시됩니다.

foo와 bar에 관한 재미있는 이야기

foo와 bar라는 용어는 프로그래밍에서 널리 사용되는 플레이스홀더 이름입니다. 이 용어는 짧고 기억하기 쉬우며 발음이 서로 달라서 많이 쓰이기 시작한 것으로 보입니다.

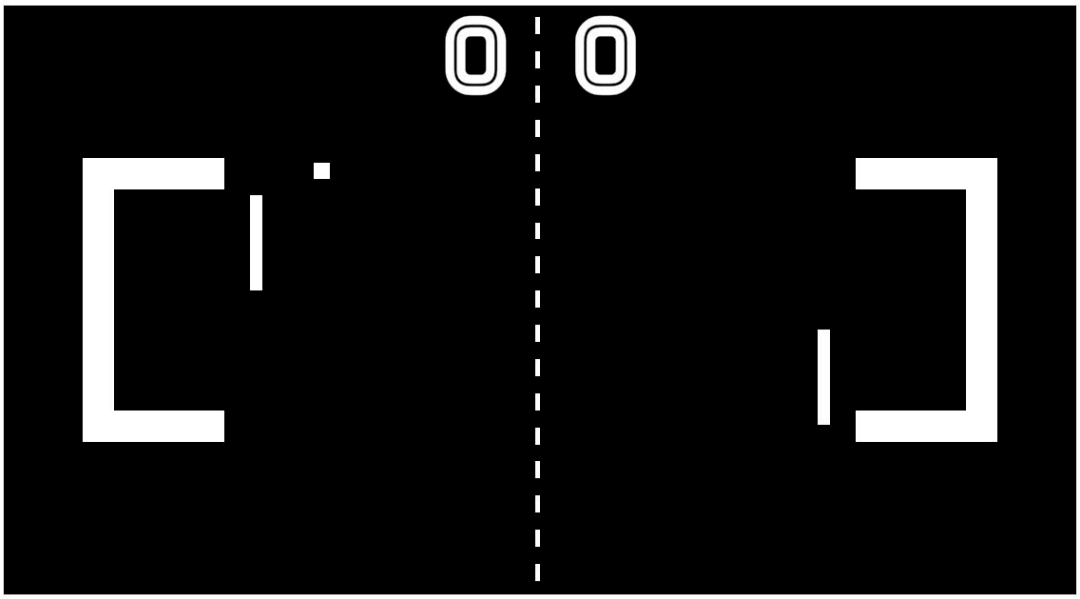
정확한 기원은 알 수 없지만, 제2차 세계대전 당시 레이더병들이 이 용어를 사용하기 시작했다는 설이 있습니다. 'foo'라는 아무 의미 없는 단어는 1930년대 만화에 캐치프레이즈로 등장하기도 했습니다.

프로그래밍 맥락에서는 1960년대에 MIT Tech Model Railroad Club이 처음으로 사용하기 시작했다고 합니다. MIT의 기차 모형실 문 옆에 'foo'와 'bar'라고 적힌 두 개의 범용 버튼이 있었는데, MIT 해커들이 이 이름을 자신들이 원하는 대로 사용하면서 일반적인 변수 이름으로 쓰이게 되었습니다. 이제는 프로그래밍 커뮤니티에서 foo와 bar를 더미 변수 이름으로 사용하는 것이 관례로 자리 잡았습니다.

샘플 프로젝트 살펴보기

이 가이드에는 스크립터블 오브젝트를 직접 다루어 보며 알아볼 수 있도록 마련된 샘플 프로젝트가 함께 제공됩니다. 고전적인 공 튀기기 게임 메카닉에서 아이디어를 얻어 탄생한 이 프로젝트는 스크립터블 오브젝트로 Unity 프로젝트의 아키텍처를 개선할 방법을 살펴보기에 아주 좋습니다.

이 프로젝트는 SOLID 원칙을 따르고 대규모 프로젝트에서 널리 사용되는 기법을 사용하므로 스크립터블 오브젝트로 코드를 재구성하는 방법을 익히기에도 적합합니다.



샘플 프로젝트

여기에서는 스크립터블 오브젝트를 적용하는 방법을 알아보고, 프로젝트의 효율과 구성을 개선하는 데 스크립터블 오브젝트가 어떤 식으로 사용될 수 있는지 살펴봅니다.

샘플에는 본 가이드에서 다룬 대부분의 패턴이 포함되어 있습니다.

— **데이터 컨테이너:** 공유 설정 데이터가 스크립터블 오브젝트로 추출됩니다. 그런 다음 게임플레이 설정 또는 공통 상태가 프로젝트에 에셋으로 저장됩니다. 스크립터블 오브젝트를 교체하여 시작 레벨의 레이아웃을 바꿀 수도 있습니다.

— **확장 가능한 열거형:** 빈 스크립터블 오브젝트가 열거형으로 작동하여 플레이어들을 카테고리로 구분합니다.

— **델리게이트 오브젝트:** 간단한 오디오 델리게이트를 사용하여 스크립터블 오브젝트를 교체하는 것만으로 공의 충돌 사운드를 무작위화하는 방법을 확인할 수 있습니다. 게임의 목표 또한 승패 조건을 정하기 위해 게임 관리 시스템에 연결되는 스크립터블 오브젝트로 구현됩니다.

— **이벤트 채널:** 관찰자 패턴을 통해 UI, 사운드, 점수 획득을 위한 게임 이벤트를 설정할 수 있습니다. 라디오 채널에 주파수를 맞추는 것처럼 여러 게임 오브젝트가 서로 다른 '이벤트 채널'을 구독할 수 있습니다.

— **이중 직렬화:** 일부 게임 데이터는 에디터에서 쉽게 사용할 수 있도록 스크립터블 오브젝트의 레벨 레이아웃처럼 저장되며, JSON 파일로 저장할 수도 있습니다. 외부에서 모딩된 JSON 데이터로 스크립터블 오브젝트를 다시 빌드할 수 있으며, 이는 원본 설정 스크립트와 연동됩니다.

이 샘플은 게임 자체에는 중점을 두지 않습니다. 공 튀기기 아케이드 게임은 이보다 훨씬 적은 코드만으로도 제작할 수 있습니다. 샘플 프로젝트의 목표는 스크립터블 오브젝트를 사용하여 쉽게 테스트하고 확장할 수 있으면서도 디자이너 친화적인 컴포넌트를 만드는 방법을 보여 주는 것입니다.

맺는말

스크립터블 오브젝트는 기존 툴셋을 보완하는 효과적인 툴이 될 수 있습니다. 디자인 패턴은 Unity 개발 환경을 구성하는 데 활용할 수 있는 다양한 가능성을 제시합니다.

본 가이드에서 다루는 여러 기법은 스크립터블 오브젝트 대신 C# 클래스를 사용해서 구현할 수도 있으며, 경우에 따라 그렇게 하기를 선호하는 개발자들도 있을 것입니다. 그러나 Unity 에디터에서는 스크립터블 오브젝트를 더 쉽고 편리하게 보고 편집할 수 있다는 사실을 잊지 마세요. 이 방식을 활용하면 코드를 통해 모든 것을 제어할 필요가 없어 아트 팀과 디자인 팀이 프로젝트 작업을 더 손쉽게 수행할 수 있습니다.

디자이너들이 소프트웨어 팀의 지속적인 도움 없이 게임플레이 데이터를 설정하고 싶어 하나요? 그렇다면 스크립터블 오브젝트가 유용한 옵션이 될 수 있습니다.

물론 패턴을 사용하지 않는 것이 패턴을 사용하는 것만큼 효과적인 경우도 있습니다. 특정 애플리케이션에는 적합한 방식이 다른 애플리케이션에는 맞지 않을 수도 있습니다. 패턴을 도입하기 전에 장점과 단점을 신중하게 살펴보세요.

Unity 프로젝트를 구성하는 방법에 적용되는 단 하나의 규칙은 없습니다. 팀원들의 역량과 개인적인 선호도를 고려하여 코드 아키텍처를 선택하세요.

그러면 게임 개발의 중요한 목표, 즉 플레이어들에게 흥미진진한 경험을 선사하는 데 집중할 수 있을 것입니다.

더 많은 자료

기술 자료

— ScriptableObject 스크립팅 API

— ScriptableObject 매뉴얼 페이지

Unity의 기술 관련 전자책

Unity의 디자인 패턴과 코딩 관련 자료

— 디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드 전자책에서는 Unity의 SOLID 개발 및 디자인 패턴을 소개합니다.

— C# 스타일 가이드로 깔끔하고 스케일링 가능한 게임 코드 작성하기(Unity 6 에디션)에서는 프로젝트 설정 베스트 프랙티스를 살펴봅니다.

그 밖의 베스트 프랙티스 가이드

— Unity 기술 자료의 고급 베스트 프랙티스 가이드

— Unity 베스트 프랙티스 허브

유나이트 자료

— MonoBehaviour의 시대에 종말을 고하는 ScriptableObject 혁명

— ScriptableObject를 사용하는 게임 아키텍처

코드 아키텍처에 관한 일반적인 정보

— 간단한 공 튀기기 게임에서 15명이 참여하는 프로젝트까지(영문)

그 밖의 프로젝트 예시

아키텍처 곳곳에서 스크립터블 오브젝트를 다양하게 사용한 커뮤니티 프로젝트 예시를 살펴보세요.

— 유나이트에서 리처드 파인이 선보인 탱크(Tanks) 데모 프레젠테이션에서는 스크립터블 오브젝트를 사용하여 오디오와 전환 가능한 AI, 파괴 가능한 건물을 커스터마이즈합니다.

— Meta의 시니어 풀스택 게임 엔지니어 라이언 히플의 GitHub 프로젝트에서는 유나이트 오스틴 프레젠테이션의 내용을 살펴볼 수 있습니다. 이와 관련된 라이언의 블로그 게시물(영문)도 읽어 보세요.

— 에셋 스토어의 SOAP - Scriptable Object 아키텍처 패턴는 본 전자책에서 설명한 다양한 패턴을 구현하며, 스크립터블 오브젝트로 작업할 때 유용한 몇 가지 기능도 함께 소개합니다.

— Unity Atoms 프로젝트는 스크립터블 오브젝트를 광범위하게 사용한 오픈 소스 라이브러리입니다. 이 페이지를 읽고 시작해 보세요.

— Reactive Menu System은 스크립터블 오브젝트를 사용하여 UI 상태 변경과 메뉴 관리를 처리하는 UGUI 시스템을 빌드합니다.

— Unity 오픈 프로젝트(Chop Chop)에서는 스크립터블 오브젝트를 적극적으로 사용하며 이벤트 채널을 사용하여 게임플레이를 관리합니다.

게임 디자이너

시스템 게임 디자인과 Unity(C#) 전문가인 시니어 테크니컬 게임 디자이너 크리스토 놉스는 Unity 게임 디자이너 플레이북 제작에 도움을 제공했습니다. 크리스토는 Unity 게임 시스템 디자인에 관한 다음 블로그 게시물 시리즈의 주요 저자이기도 합니다.

— 생태계를 만드는 시스템: 새로운 게임 디자인

— 예측할 수 없는 즐거움: 무작위화가 게임 설계에 주는 이점

— 애니메이션 커브를 최고의 디자인 레버로 활용하는 방법

프로페셔널 트레이닝

Unity 프로페셔널 트레이닝은 팀원들이 기술과 지식을 쌓고 Unity에서 더 생산적으로 작업하며 효율적으로 협업하는 데 도움이 되는 온라인 및 대면 교육을 제공합니다. 여기에서 Unity 프로페셔널 트레이닝에 대해 자세히 알아보세요.

unity.com/kr