UNITY
-CAHPTER2-

SOUL SEEK

목차

1. 동작원리를 이해하기 위한 지식
2. 유니티에서의 컴포넌트
3. 메시지와 브로드 캐스팅
4. Script생성과 사용

동작원리를 이해하기 위한 지식

1. 동작원리를 이해하기 위한 지식

상속과 재사용

엔진이라는 도구를 사용하기 위해서는 이미 만들어진 기반코드를 재사용하는 방법의 이해가 필요하다.
유니티의 컴포넌트 기반 구조를 이해하려면 코드를 재사용하는 전통적인 방법인 '상속'을 알아야 한다. ➔ 이미 만들어진 클래스에 새로운 코드와 기능을 덧붙여 새로운 클래스를 만드는 개념

상속의 이해를 돕기 위해 몬스터를 만들어보자.

```
class Monster
{
    // 몬스터에 대한
    // 변수와 메서드들..
}
```

오크**(Orc)**와 오크대장**(Orc Chieftan)**를 만든다고 했을때..

Class Monster, class Orc : Monster, class OrcChieftan : Orc 이렇게 각각 상속관계를 만들어 사용할 수 있다.
➔ 오크와 오크 대장만 만들어도 되지만 **Monter**를 만든것은 오크 이외의 몬스터들의 특징도 포함하기 위해서 이다.

1. 동작원리를 이해하기 위한 지식

Monster 클래스는 몬스터로서 필요한 다음 필수 기능을 가지고 있다.

인공지능 기능, 애니메이션 기능, 공격과 방어 기능, 물리 기능, 기타 필수 기능

보는 것과 같이 몬스터라는 녀석은 **Enemy**속성을 가진 **Data**부분을 다루기 위한 것이다. 사실 몬스터로 다루어 지는 **Enemy**속성의 오브젝트들은 결국 외형에서 보여지는 부분으로 구분하지만 **Data**로서는 다 같은 놈이고 큰 동작 구분만 가지고 있다면 모든 **Enemy**는 몬스터를 상속하게 된다.

class Orc : Monster 부모의 클래스를 상속하면서 이미 부모가 가지고 있는 기능을 사용하거나 **Override**해서 사용하고 나머지 부분은 구현하자.

초록색 피부, 오크의 애니메이션, 오크의 스킬, 그외 오크의 고유 기능

class OrcChieftan : Orc 오크가 가진 기능에 더해진 일반 오크와 차별화 된 것을 추가 할 것이기 때문에 그렇게 많은 구현이 필요하진 않다.

대장모자, 새로운 무기와 강력한 스킬, 그외 오크 대장의 고유 기능

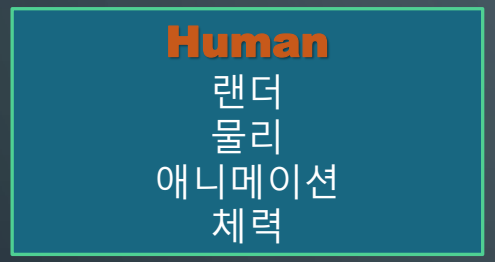
1. 동작원리를 이해하기 위한 지식

상속의 한계

부모 클래스를 상속해 자식 클래스의 기초 구현을 대신할 수 있다. 하지만 상속에만 의존하면 오히려 코드를 재사용하기 힘들 수 있다.

RPG게임에서 플레이어와 **NPC,** 몬스터를 만들어보자.

최상위 부모 클래스인 Human 클래스



Human 클래스는 사람 형태를 가진 클래스의 부모 클래스로 사용한다.
사람 형태를 가진 오브젝트에 필요한 기능을 미리 예상해서 **Human** 클래스에 추가한다.
모습을 그려주는 랜더 기능, 물리기능, 애니메이션 기능, 체력 기능, 기타 필수 기능

플레이어 클래스 구현

플레이어사 직접 조작하는 캐릭터인 **Player** 클래스를 만드는데 **Player**클래스는 **Human**클래스를 상속하여 **Human**의 모든 기능을 가진다.

조작 기능, 공격 기능, 직업 기능, 기타 필수 기능

1. 동작원리를 이해하기 위한 지식

이제 **NPC**와 **Monster**를 **Human** 클래스를 이용해서 만들어 보자.

NPC는 플레이어와 대화, 거래 등의 상호작용을 하기 때문에 이 같은 기능들이 추가 된다.

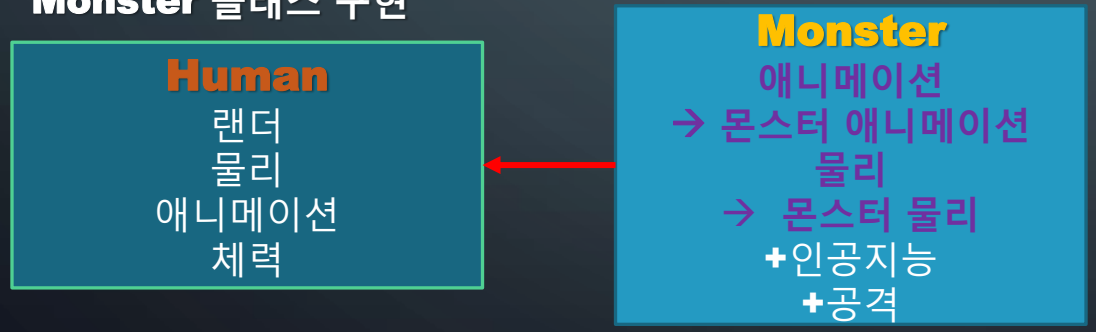
NPC 클래스 구현

NPC는 물리와 체력기능이 필요 없는 경우가 많다.
NPC는 상호작용을 위한 구성이라면 물리작용도 딱히 필요가 없다.
결국 **Human** 클래스로부터 물려받은 체력과 물리 기능을 제거하고 제거한 기능 관련된 다른 기능에 에러가 발생하지 않도록 코드를 정리하는 추가 작업도 해야 한다.

상속을 이용하면서 **NPC**는 오히려 추가 작업이 생겼다..

Monster 클래스는 **Human**의 거의 모든 기능을 바꿔야 하는 일이 발생한다.

Monster 클래스 구현



애니메이션이나 물리의 경우 일반적인 플레이어라면 이족보행의 경우가 많지만 몬스터는 항상 그런 것이 아니기 때문에 애니메이션 구현을 같이 쓸 수 없다. 물리 또한 **Monster**에 작용하는 물리가 **Player**랑 다른 경우가 많다.**(ex** 날아다니는 **Monster)**
Monster와 어울리지 않는 부분은 수정 또는 삭제를 해서 사용해야하는 상황이 되었다.

1. 동작원리를 이해하기 위한 지식

결론,

상속은 재사용에 있어서 훌륭한 기능을 지원하고 있지만 우리가 실제 게임을 만들려고 하는 상황에 따라서는 불편한 부분이 있는 것은 사실.

부모 클래스에는 자식 클래스에 공통적으로 필요한 기능을 구현한다. 그런데 나중에 구현할 자식 클래스에 무엇이 필요한지 처음부터 정확하게 추측하기 힘들다.
또한 부모 클래스의 기존 기능이 나중에 구현한 자식 클래스의 기능과 오히려 충돌할 수 있다.

이외에도 상속에만 의존하면 기획자가 새로운 오브젝트를 만들 때 매번 프로그래머에게 부탁해야 하는 문제가 생긴다. 프로그래머만이 부모 클래스를 확장하여 새로운 자식 클래스를 만들 수 있기 때문이다.

문제점을 요약하면.

- **오히려 코드를 재사용하기 힘든 경우가 생길 수 있다.**
- **새로운 오브젝트를 만들려면 프로그래머에게 의존해야 한다.**

이러한 문제들을 해결하기 위한 도구가 **컴퍼넌트 패턴**이다.

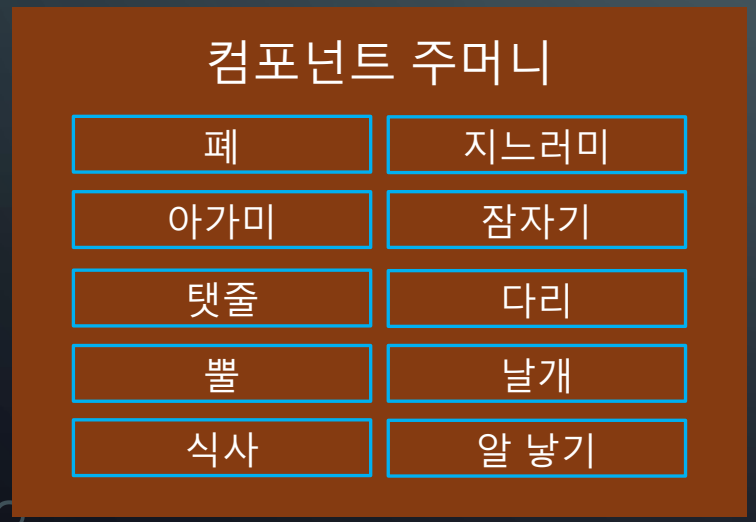
1. 동작원리를 이해하기 위한 지식

컴포넌트 패턴

- 컴포넌트 혹은 컴포지션 패턴이라고 부른다.
- 미리 만들어진 부품을 조립하여 완성된 오브젝트를 만드는 방식이다.
  ➔미리 만들어진 부품을 컴포넌트라 부르며 컴포넌트마다의 대표기능을 가지고 있다.
- **컴포넌트 패턴에서 게임 오브젝트는 속이 빈 껍데기**이다. 개발자는 빈 게임 오브젝트에 컴포넌트를 조립하여 새로운 기능을 추가할 수 있다.

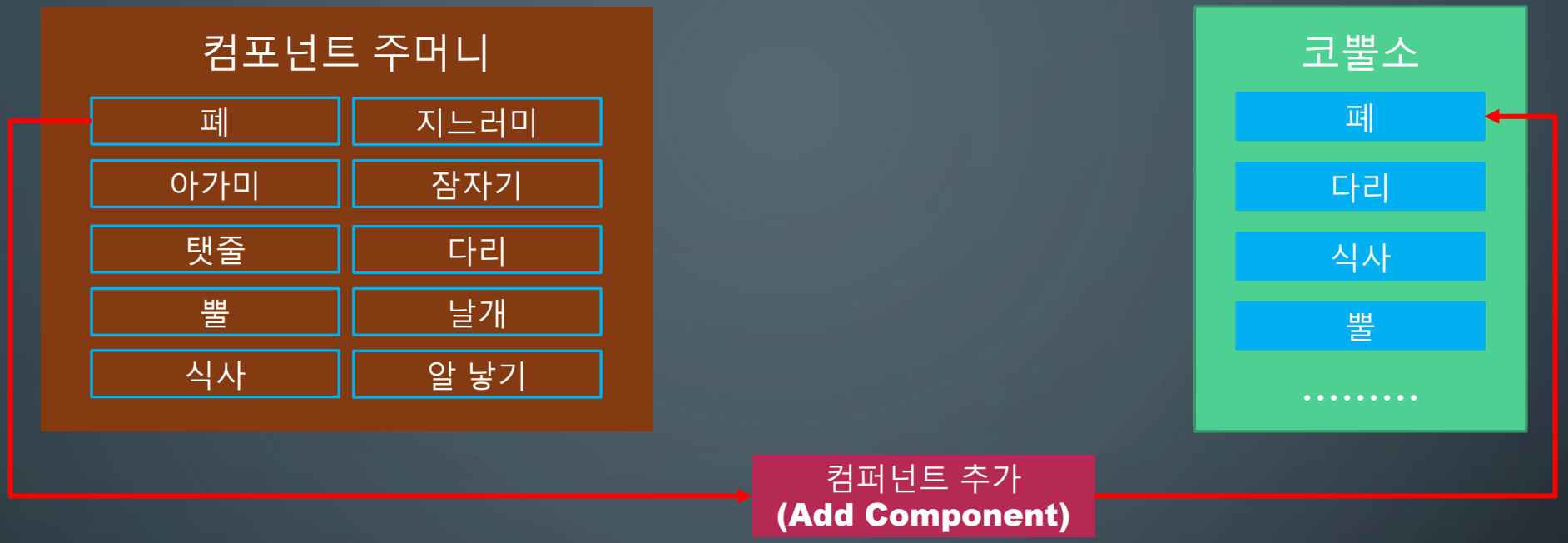
동물이 나오는 게임을 만들자고 했을 때..

미리 여러 기능하는 것들을 만들어 두고 이것을 조립해서 사용할 수 있게 하자.



코뿔소라는 동물을 만들고자 했을 때 준비된 컴포넌트들을 붙일 수 있는 뼈대나 홀더 역할을 하는 것을 만들어 두면 된다.
그러면 사용하고 싶은 기능을 붙여서 사용하기만 하면 되는 것이다.

1. 동작원리를 이해하기 위한 지식



게임 오브젝트와 컴포넌트의 특징

미리 만들어진 컴포넌트를 빈 껍데기인 게임 오브젝트에 조립하는 방식이라는 특징을 가지고 이런 특징이 가져오는 장점이 있다.

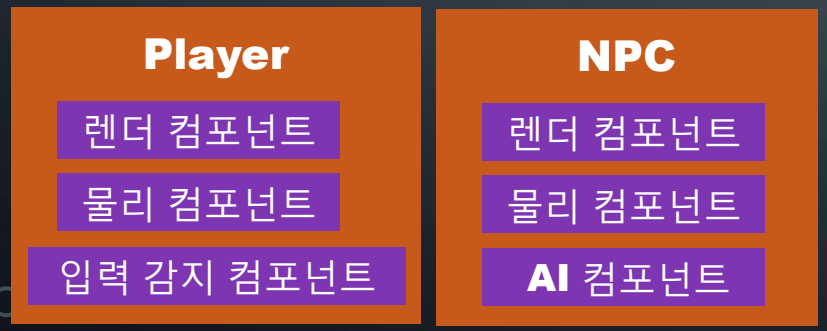
- 유연한 재사용이 가능하다.
  ➔상속만을 사용한 경우에서는 부모 클래스의 불필요한 기능까지 모두 가져오기 때문에 코드 재사용이 힘든 경우가 많다. 컴포넌트에서는 원하는 기능을 가진 컴포넌트만 선택적으로 골라 쓸 수 있다.
- **독립성** 덕분에 기능 추가와 삭제가 쉽다.
  ➔겉으로는 간단해 보이는 수정사항에도 프로그래머가 두려워하는 모습을 가끔 볼 수 있다. 코드의 한 부분만 수정하더라도 관련된 여러 부분의 코드가 망가질 수 있기 때문이다. 하지만 컴포넌트 방식에서는 어떤 기능을 추가하거나 삭제할 때 다른 기능이 망가지지 않기 때문에 그런 걱정이 줄어든다.

1. 동작원리를 이해하기 위한 지식

컴포넌트의 독립성

독립성이라는 강력한 특징은 두가지 특징에서 파생된다.
게임 오브젝트는 단순한 빈 껍데기이다.
➔몇 가지 식별 기능과 자신에게 어떠한 컴포넌트가 조립되어 있는지 알 수 있는 기능을 제외하면 특별한 기능은 없다.
컴포넌트는 스스로 동작하는 독립적인 부품
컴포넌트는 자신과 같은 게임 오브젝트에 추가된 다른 컴포넌트에 관심이 없다. 컴포넌트의 기능은 컴포넌트 내부에 완성(완결)되어 있기 때문이다. 그러므로 컴포넌트는 다른 컴포넌트에 의존하지 않는다.
즉, 게임 오브젝트에 어떤 컴포넌트를 마음대로 조립하거나 빼도 다른 컴포넌트가 망가지지 않는다.

현재 **Player** 게임 오브젝트와 **NPC** 게임 오브젝트는 외형을 그려주는 랜더 컴포넌트와 물리 컴포넌트를 가지고 있다. 이때 **Player**나 **NPC**에서 물리 컴포넌트를 제거하면 물리 기능만 사라진다는 점에 주목하면 된다. 물리 컴포넌트가 제거 되더라도 게임 오브젝트 자체는 전혀 망가지지 않는다.



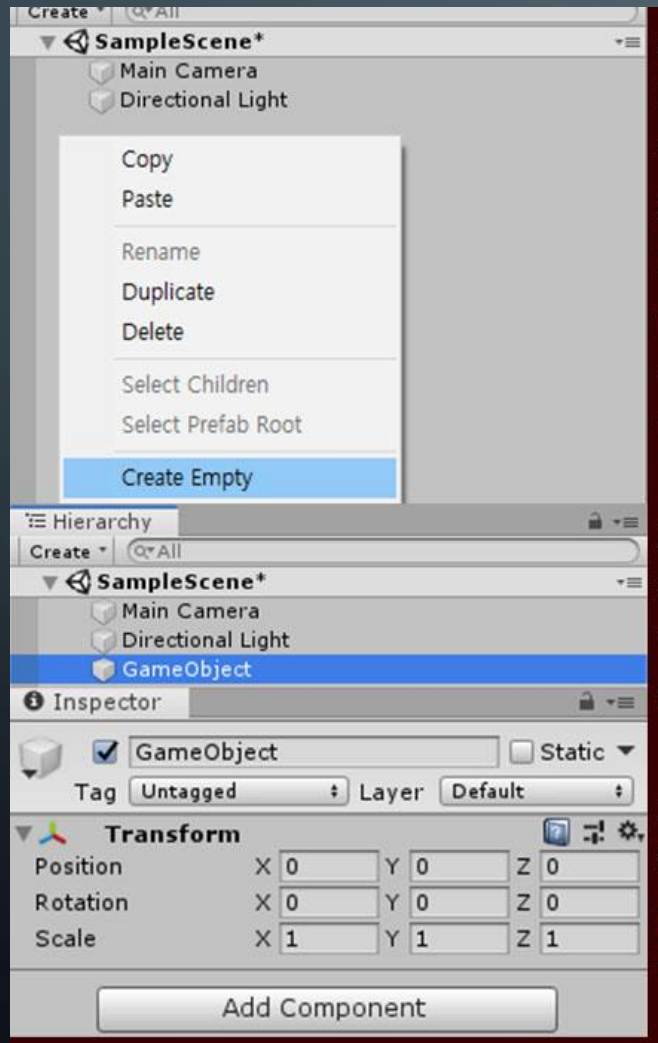
각자의 새로운 기능을 가진 컴포넌트를 추가했지만 기존 컴포넌트를 수정할 필요는 없다. 결론적으로 컴포넌트 방식에서는 새로운 기능을 추가하거나 삭제할 때 기존 기능이 망가질까 걱정할 필요가 전혀 없다.

유니티에서의 컴퍼넌트

2. 유니티에서의 컴퍼넌트

GameObject는 컴포넌트를 **Scene**상에 나타나게 하기 위한 껍데기
Scene에서 컴포넌트들이 표현되어야 한다면 **GameObject**라는 껍데기가 반드시 있어야한다.

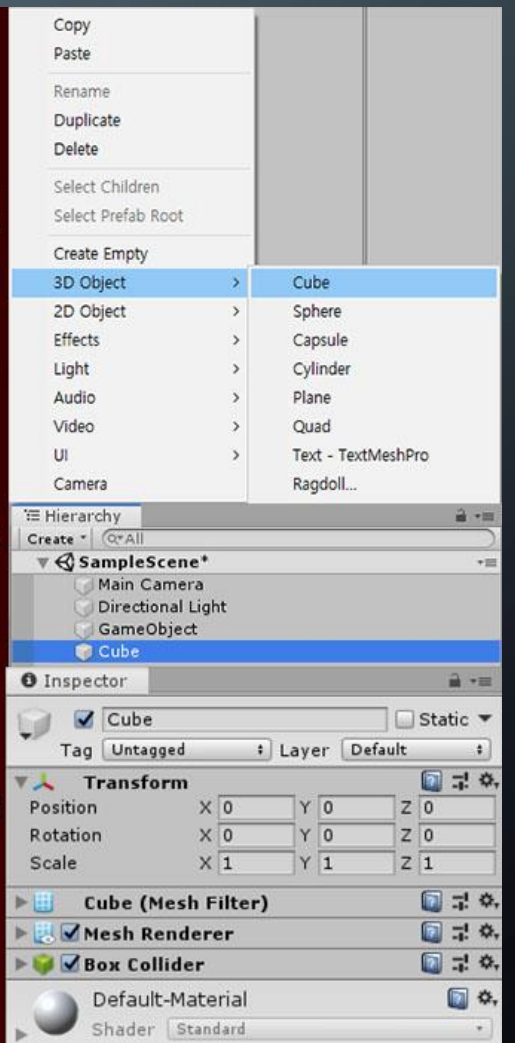
GameObject 생성



하이어라이키 창에서 우클릭을 한뒤 나오는 팝업 메뉴에서 Create Empty를 클릭하면 빈 껍데기만 있는 GameObject가 생성된다.
공간에 존재해야 하기 때문에 Transform 컴퍼넌트는 가지고 생성된다. 아래에 있는 Add Component 버튼으로 Asset 폴더 내에 있는 Component들을 추가 할 수 있다. 우리가 앞으로 해야할 일들이다.

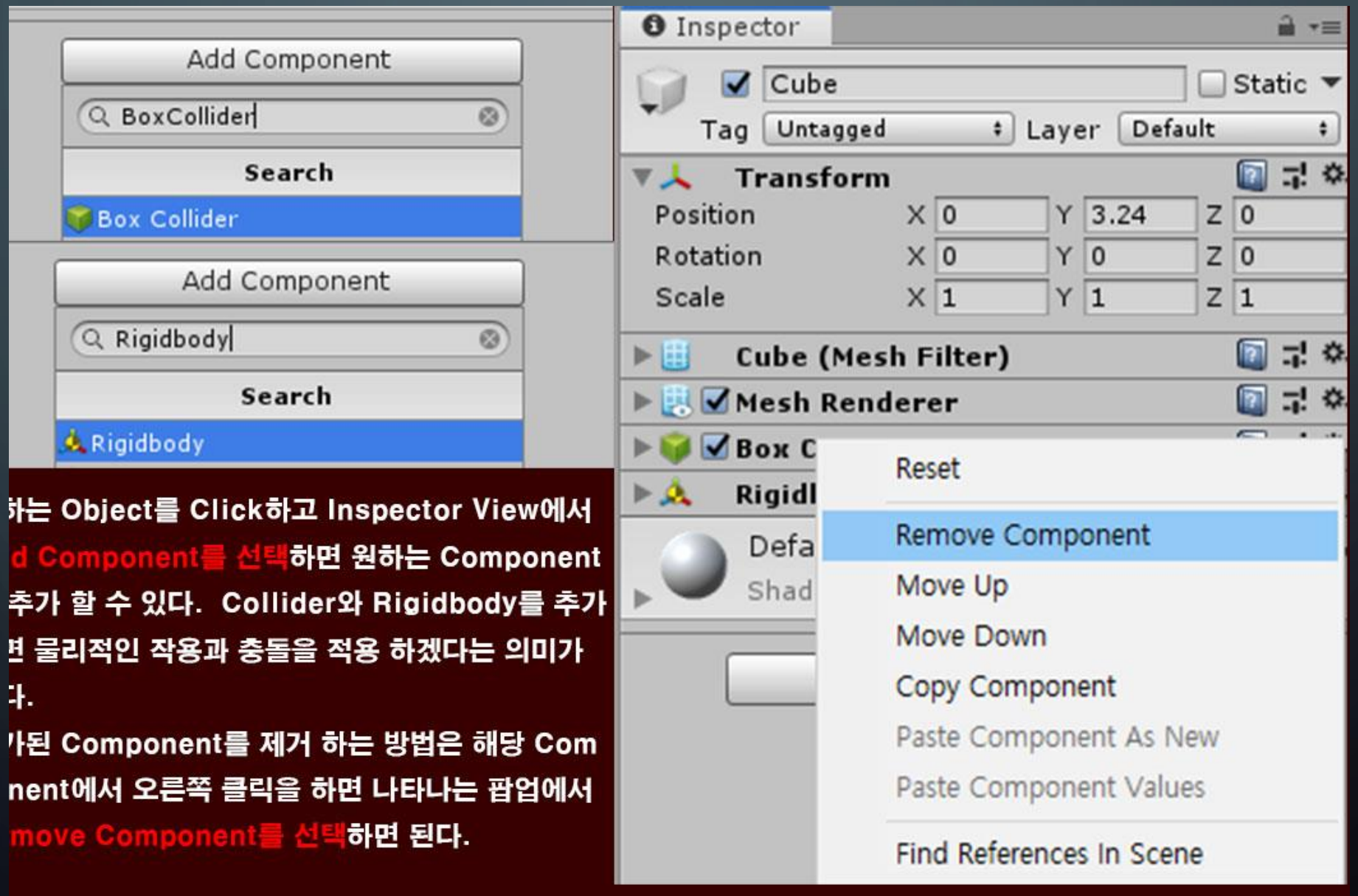
그 이외에 Unity에서 이미 제공하고 있는 Object 셋들이 있다 3D, 2D, Effect, Light, Audio, Video, UI, Camera가 있다. 그중 Cube를 선택해서 보면 잔뜩 컴퍼넌트 된 상태를 볼 수 있다.

Box Collider : 다른 물체가 부딪칭수 있는 물리적인 표면을 만듭니다. 물체와 물체의 충돌은 Collider가 체크 한다. 물리적인 충돌을 일으킬려면 Collider가 있어야 한다.



2. 유니티에서의 컴퍼넌트

Component추가 및 삭제

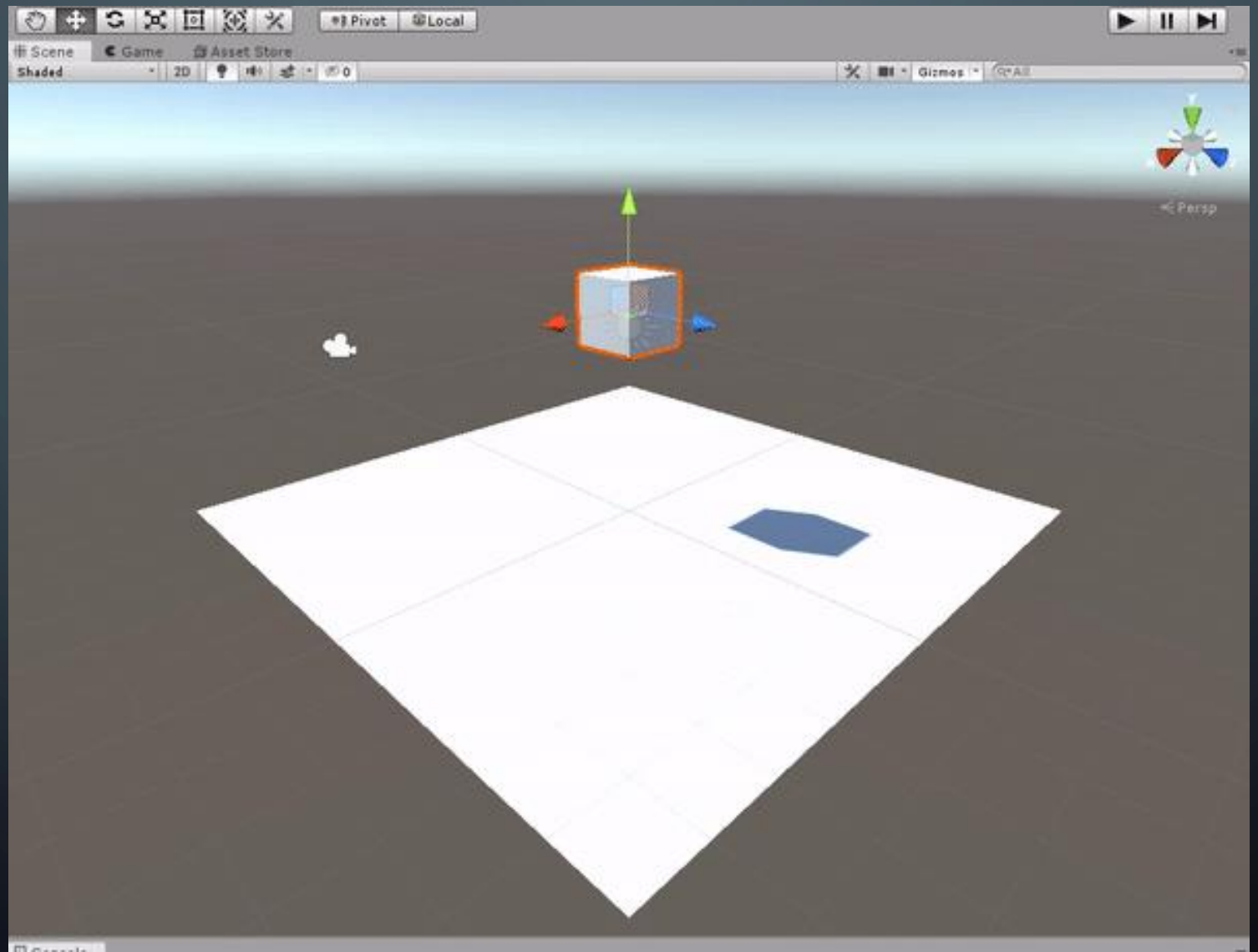


하는 Object를 Click하고 Inspector View에서
d Component를 선택하면 원하는 Component
추가 할 수 있다. Collider와 Rigidbody를 추가
면 물리적인 작용과 충돌을 적용 하겠다는 의미가
다.

가된 Component를 제거 하는 방법은 해당 Com
nent에서 오른쪽 클릭을 하면 나타나는 팝업에서
move Component를 선택하면 된다.

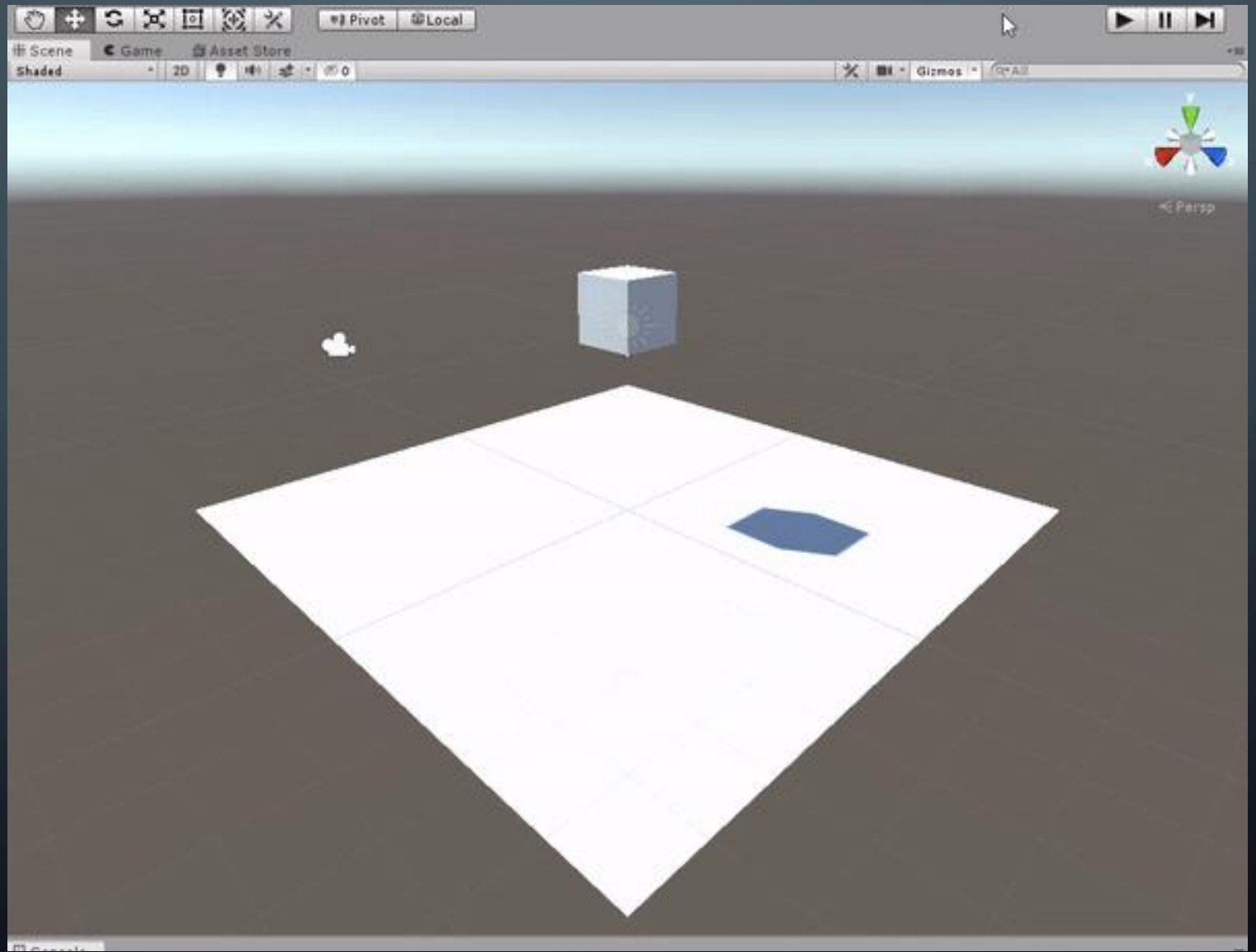
2. 유니티에서의 컴퍼넌트

Cube와 **Plan**을 생성해서 **Play**를 시켜보면 **Collider**가 기본적인 적용으로 되어 있기 때문에 두 **Object**가 충돌하면 멈추게 된다**.**



2. 유니티에서의 컴퍼넌트

반대로 충돌을 체크하는 **Collider**를 제거하면 서로 충돌을 하지 않고 물리 작용만 적용 받는다**.** 기본적으로 **Rigidbody**는 중력 값 **y**방향으로 **-9.8**을 적용 받고 있는 상태이다**.**



메시지 브로드캐스팅

4. 메시지 브로드캐스팅

컴포넌트 구조에서는 '전체 방송'을 이용해 컴포넌트의 특정 기능을 간접적으로 실행할 수 있다. 이러한 전체 방송을 '브로드캐스팅'이라 부른다.

MonoBehaviour

- **Unity**의 모든 **Component**들은 **MonoBehaviour** 클래스를 상속한다.
- **Unity**에서 미리 만들어 제공하는 클래스이며 컴포넌트에 필요한 기본 기능을 제공한다. 즉, **Monobehaviour**를 상속한 클래스는 **GameObject**에**Component**로 추가될 수 있다.
- **Monobehaviour**를 상속해서 만든 **Component**는 유니티의 제어를 받게 되므로 **Component**는 **Unity**이 메시지를 들을 수 있다.

Unity의 메시지 기반

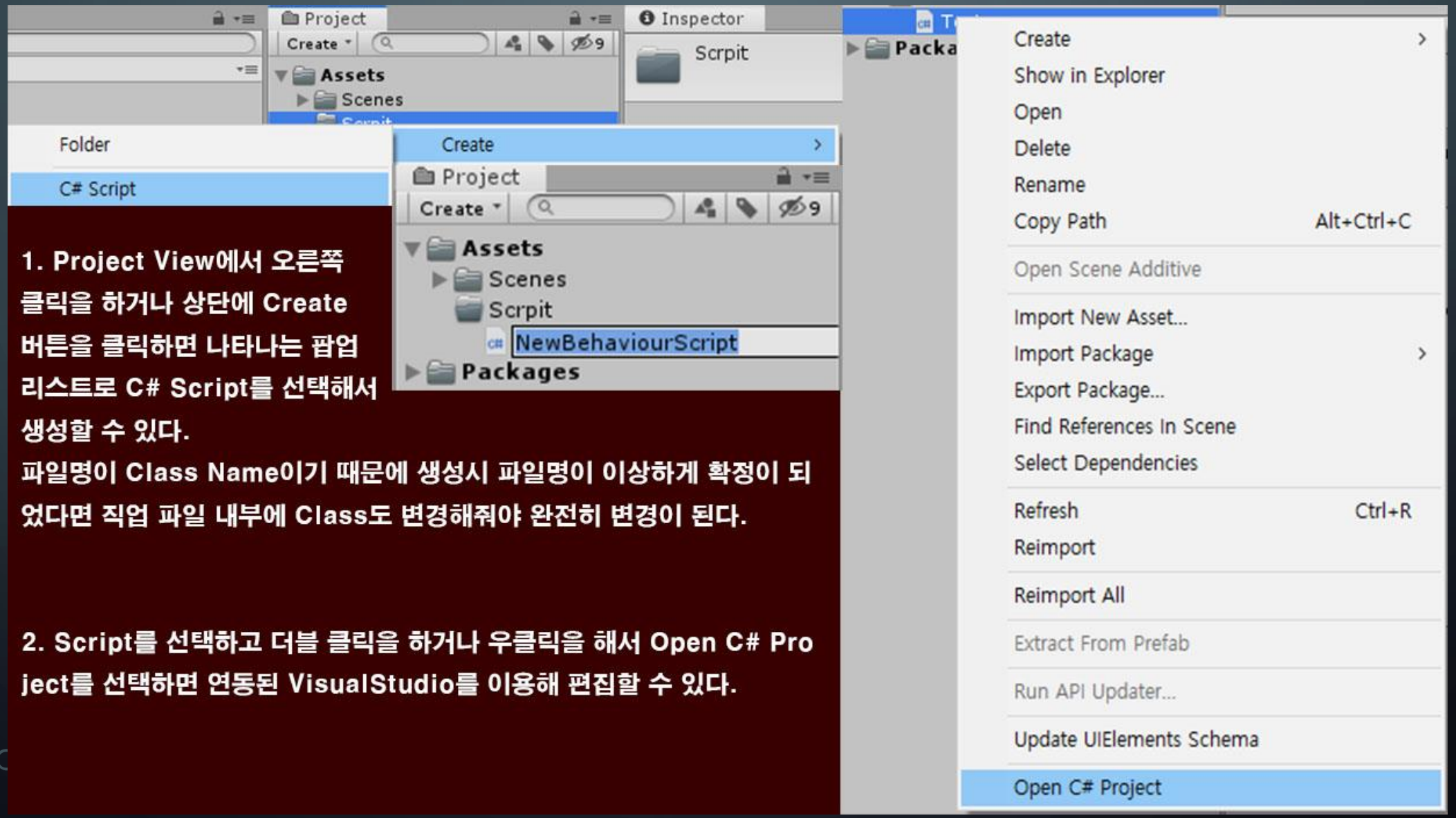
- 컴포넌트패턴에서 컴포넌트들은 서로에게 관심이 없다. 필요에 의해 찾기 전까지는 서로의 존재를 알 수 없다.

- **Unity**에서 **Monobehaviour**를 통해 어떤 일을 실행하고 싶을 때, 당사자를 직접 찾아서 명령을 내리는 방법을 선택했다.
  ➔모든 **Component**에 이를 전달한다면 상당히 비효율적이다. 그래서 **Monobehaviour**는 자신을 상속한 **Component**들이 활성화(**Scene**에 존재하는지)하는지 체크하고 각 상속해서 제공된 **Method**가 존재할 경우에 그 **Method**의 실행으로 메시지를 전달하게 된다. 상속 **Method**가 아닌 다른 **Method**의 경우에는 개발자가 직접 해당 **Object**를 찾아서 **Component**를 할당 받아서 사용해야 한다.

SCRIPT의 생성과 사용

4. SCRIPT의 생성과 사용

Script 생성



1. Project View에서 오른쪽 클릭을 하거나 상단에 Create 버튼을 클릭하면 나타나는 팝업 리스트로 C# Script를 선택해서 생성할 수 있다.
파일명이 Class Name이기 때문에 생성시 파일명이 이상하게 확정이 되었다면 직업 파일 내부에 Class도 변경해줘야 완전히 변경이 된다.

2. Script를 선택하고 더블 클릭을 하거나 우클릭을 해서 Open C# Project를 선택하면 연동된 VisualStudio를 이용해 편집할 수 있다.

4. SCRIPT의 생성과 사용

Script 사용

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

public class Test : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {

    }
}
```

생성된 **Script**는 **GameObject**라는 빈 껍데기에 역할을 부여한 **Component**가 되는 것이다 그렇기 때문에 기본적으로 **MonoBehaviour**를 상속받고 최초 오브젝트가 활성화 되는 순간 실행 되는 **Start Method**와 매 프레임 마다 실행되는 **Update Method**를 가지고 있다.

해당 **Method**들은 필수 요건이 아니며 필요성에 따라 생략할 수 있고 필요에 따라 기타 여러가지 **Method**들을 사용할 수 있다.

4. SCRIPT의 생성과 사용

Monobehaviour에 의해 자동으로 관리되는 메시지 Method

- **Awake()**
  - 스크립트 실행 시 한번 실행, 스크립트가 비활성화 되어 있어도 실행
  - 주로 게임의 상태 값 또는 변수 초기화에 사용
  - **Start**함수가 호출되기전에 맨 먼저 호출된다.
  - **Corutine** 실행 불가.

- **Start()**
  - **Update** 함수가 호출되기전 한번만 호출
  - **Awake**가 끝난 뒤 실행, **Corutine** 실행가능

- **Update()**
  - 매 프레임마다 호출

- **LateUpdate()**
  - 모든 업데이트가 이루어진 후 한번씩 호출한다.
  - 카메라 업데이트에 주로 사용한다.

- **OnEnable(), OnDisable()**
  - 이벤트 연결 / 연결종료 시 사용.

4. SCRIPT의 생성과 사용

- **FixedUpdate()**
  - 고정된 시간마다 호출되는 함수**(1**초에 **50fps** 환경에서 **0.2**초마다 호출**)**
- **OnPreRender(), OnRenderObject(), OnPostRender()**
  - **Scene** 렌더링 전**,** 렌더링 후**,** 모든 렌더링이 끝난 후 호출되는 함수
- **OnGUI()**
  - **GUI** 이벤트에 대한 응답으로 프레임당 여러번 붙려진다**.**
- **OnDrawGizmos()**
  - 시각화 목적으로 씬뷰에서 기즈모를 그리기 위해 사용된다**.**
- **OnApplicationQuit()**
  - 어플리케이션이 종료하기 전에 호출되는 함수**.**